TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TUOMAS SALMI

LAAJENNETTAVAN ARKKITEHTUURIRUNGON TOTEUTUS LUONNOLLISEN KIELEN TIEDONLOUHINTAAN

Diplomityö

Tarkastaja: Prof. Hannu-Matti Järvinen

Tarkastaja ja aihe hyväksytty 1. marraskuuta 2017

TIIVISTELMÄ

TUOMAS SALMI: Laajennettavan arkkitehtuurirungon toteutus luonnollisen kielen tiedonlouhintaan
Tampereen teknillinen yliopisto
Diplomityö, 50 sivua
Marraskuu 2018
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Pervasive Systems
Tarkastaja: Prof. Hannu-Matti Järvinen

Avainsanat: ohjelmistoarkkitehtuurit, luonnollisen kielen käsittely, tiedonlouhinta, tietovuo, plugin-arkkitehtuuri

Suomen sotiin liittyen on Suomessa 1970-luvulla koottu useita matrikkelikirjasarjoja, jotka sisältävät tietoa sotiin liittyneistä ihmisryhmistä. Merkittävä tällainen ihmisryhmä on luovutetusta Karjalasta evakkoon lähteneet siirtokarjalaiset.

1970-luvulta peräisin olevat paperiset kirjat on aikaisemmin digitoitu PDF-muotoon ja ne sisältävät tieteellisesti kiinnostavaa dataa esimerkiksi yhteiskuntatieteellistä tai biologista tutkimusta varten. Tässä työssä esitellään sovelluskehys, joka kehitettiin Helsingin yliopiston tutkimushankkeessa louhimaan henkilötietoja digitoiduista kirjasarjoista tieteellistä tutkimusta varten.

Sovelluskehyksen suunnittelussa korostettiin laajennettavuutta ja sen soveltamiskelpoisuutta useisiin tutkimusprojektille merkityksellisiin kirjasarjoihin. Työn tuloksena kehitettiin *Kaira*-sovelluskehys, joka mahdollistaa erilaisten tiedonlouhinta-algoritmien kokoamisen yhteen ajettavaan sovelluskehykseen. *Kaira* tukee erityisesti kirjasarjojen louhintalogiikan laajentamista sekä helpottaa tiedonlouhinta-algoritmien kirjoittamista uusille kirjasarjoille. Sovelluskehyksen ensisijaisiksi arkkitehtuuriratkaisuiksi valikoituivat tietovuoarkkitehtuuri ja plugin-arkkitehtuuri.

ABSTRACT

TUOMAS SALMI: The implementation of an extensible software framework for natural language data mining
Tampere University of Technology
Master of Science thesis, 50 pages
November 2018
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Professor Hannu-Matti Järvinen

Keywords: Software architecture, natural language, data mining, data flow, plugin architecture

In the 1970s, several catalogues related to the populations who were involved in the Finnish wars were collected. One significant group of people was the Karelians, who were forced to migrate from Southern Karelia during the Soviet occupation of the area.

The original paper books from the 1970s were digitized into PDF-format. They contain interesting data for researchers in the fields of biology and sociology. A software framework developed for mining personal data for research purposes from the digitized books is introduced in this work. The framework was developed as part of a University of Helsinki research project.

The design of the framework focused on general extensibility and its applicability to multiple book series deemed important by the research project. The end result was the *Kaira* framework, which enables one to compose several datamining algorithms in one executable application. *Kaira* supports expanding the datamining logic of the previously supported book series and eases the development of new algorithms for previously unsupported book series. The foremost architecture solutions for the framework were data flow architecture and plugin architecture.

ALKUSANAT

Tämä diplomityö syntyi osana Helsingin Yliopiston *Learning from our past* tutkimushanketta, johon osallistuin ohjelmistokehittäjän roolissa. Työssä kuvattu sovellus sai alkunsa jo vuonna 2015 suorittaessani siviilipalvelusta Lammin biologisella asemalla, jossa John Loehr esitteli minulle aiheen ja kiinnostukseni heräsi kehittää sovelluskehys tutkimuskäyttöön. Lämpimät kiitokset siten Johnille työn alkuperäisestä ideasta ja mahdollisuudesta kehittää sovellusta pidemmälle osana myöhemmin käynnistynyttä tutkimusprojektia. Tutkimusprojektin henkilökunnasta tahdon kiittää myös Juuso Kallioniemeä käytännön ohjelmointityöstä, mikä helpotti omaa työtaakkaani diplomityön kirjoittamisen aikana.

Tahdon kiittää myös kaikkia diplomityöni kirjoitusprosessiin osallistuneita: työni tarkastajaa ja ohjaajaa Hannu-Matti Järvistä ja Tuukka Vainiota, Enni Salmea ja Antti Pulakkaa palautteesta ja parannusehdotuksista kirjoitustyön aikana. Kiitokset myös työnantajalleni Vincitille, joka mahdollisti diplomityön tekemiselle oleellisen joustavan työsuhteen.

Tampereella, 3.11.2018

Tuomas Salmi

SISÄLLYSLUETTELO

KUVALUETTELO

LYHENTEET JA MERKINNÄT

ADD	*Attribute driven design.* Arkkitehtuurisuunnittelumenetelmä, jossa arkkitehtuurin rakenne hahmotellaan skenaarioista löydettyjen toiminnallisten vaatimusten ja laatuvaatimusten pohjalta.
ALMA	*architecture level modifiability analysis* Menetelmä ohjelmistoprojektin kustannusten ja riskien ennustamiseen.
ATAM	*architecture tradeoff analysis method* Menetelmä ohjelmistoarkkitehtuurin kattavaan arviointiin. Arvioi kuinka hyvin arkkitehtuuri tyydyttää laatutarpeet sekä analysoi laatuvaatimusten suhteita toisiinsa.
chunk ja "chunking"	Prosessi, jossa kirjasarjan tekstin sisältävä html-tiedosto muunnetaan analysoitavissa olevaan XML-formaattiin, joka koostuu yksittäisen henkilötiedon sisältävistä elementeistä. Raakateksti jaetaan siten "kimpaleiksi"(chunk).
ConNLL-U	Formaatti luonnollisen kielen lauserakenteiden ja sanojen suhteiden kuvaamiseen.
DCAR	*Decision-centric architecture review method* Kevyt ja iteratiivinen ohjelmistoarkkitehtuurin arviointimenetelmä.
Ensisijainen henkilö	Kirjasarjan tekstikappaleen "nimikkohenkilö", jota tekstikappale ensisijaisesti kuvaa. Tekstikappaleessa voi olla mainintoja muista luonnollisista henkilöistä, esimerkiksi ensisijaisen henkilön puolisosta.
Evolutiivinen arkkitehtuurisuunnittelu	Suunnittelumenetelmä, jossa koko arkkitehtuuria ei suunnitella pitkälle ennakkoon. Suunnittelupäätöksiä tehdään projektin edetessä evolutiivisesti. Menetelmä luottaa erityisesti refaktorointiin, testaamiseen ja arkkitehtuurin jatkuvaan uudelleenarviointiin kehitystyön aikana.

Hollywood-periaate	*"Don't call us, we'll call you"* Suunnitteluratkaisu, jossa sovelluskehyksestä erikoistettu ohjelmistomoduuli ei ohjaa sovelluksen suoritusta vaan sen sijaan sovelluskehys kutsuu sitä tarvittaessa.
HTML	*Hypertext Markup Language* on kuvauskieli hyperlinkkejä sisältävien tekstidokumenttien määrittäseen. HTML-dokumentit voivat viitata toisiinsa hyperlinkkien avulla. HTML-dokumentteja käytetään esimerkiksi internetsivujen rakentamiseen.
Louhintakomponentti	Ohjelmistokomponentti, jonka tehtävänä on irrottaa kirjasarjan tekstikappaleesta jokin yksittäinen datan kappale hyödyntäen tehtävään suunniteltua algoritmia.
Matrikkeli	Henkilötietohakemisto jostakin ihmisryhmästä tai organisaatiosta.
Muunneltavuustaktiikka	Ohjelmistosuunnittelutaktiikan tyyppi, joka pyrkii parantamaan ohjelmiston muunneltavuutta myöhemmin.
Muutoskäyrä	Käsite, joka kuvaa ohjelmistoon tehtävien muutosten hinnan kasvua projektin edetessä. Usein arvioidaan eksponentiaalisesti kasvavaksi.
OCR	*Optical Character Recognition*, tekstintunnistusteknologia.
Refaktorointi	Olemassaolevan ohjelmakoodin rakenteen muokkaamista muuttamatta sen toiminnallisuutta.
Regressiotestaus	Testauskäytäntö, jossa sovellusta testataan toistuvasti vanhaan koodiin ilmestyneiden mahdollisten virheiden löytämiseksi. Tavallisesti toteutettu automaattitestauksella.
Suunnittelumalli	Yleisesti hyväksi havaittu ratkaisu yleiseen ohjelmistosuunnittelussa kohdattavaan ongelmaan. Voidaan katsoa koostuvan joukosta yksittäisiä taktiikoita.

Taktiikka	Suunnittelupäätös, joka mahdollistaa ohjelmiston laatuvaatimusten täyttämisen.
Tuoterunko-arkkitehtuuri	Ohjelmistoarkkitehtuuri, joka mahdollistaa ohjelmakoodin systemaattisen jakamisen tuoteperheen sovellusten välillä.
Värevaikutus	Vaikutus koodikannan muihin osiin, kun jotain sen osaa muokataan. Muutos tiettyyn paikkaan voi aiheuttaa muutostarpeita muihin paikkoihin, jolloin muutostarve väreilee muualle.
XML	*Extensible Markup Language* on merkintäkielistandardi, jota käytetään tiedon rakenteelliseen esittämiseen ja tallentamiseen koneluettavassa muodossa.

1. JOHDANTO

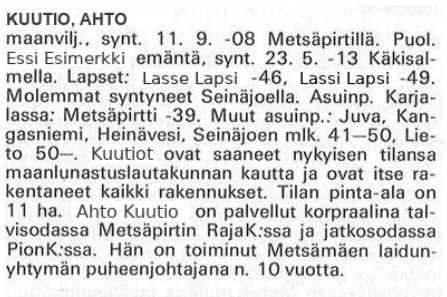
Tässä diplomityössä käsitellään *Kaira* [24] sovelluskehyksen arkkitehtuuria. Sovelluskehyksen tehtävänä on helpottaa tiedon louhimista luonnollisella kielellä kirjoitetuista suomalaisista matrikkelikirjasarjoista.

70-luvulla Suomessa koottiin useita matrikkelikirjasarjoja, jotka sisältävät henkilötietoja esimerkiksi Suomen sotiin osallistuneista henkilöistä sekä luovutetun Karjalan entisestä väestöstä. Nämä kirjasarjat sisältävät lyhyitä henkilökuvauksia kymmenistä tuhansista ihmisistä, ja ovat siten potentiaalinen tiedon lähde sosiaaliselle, antropologiselle ja biologiselle tutkimukselle. Tilastollisen tutkimustyön mahdollistamiseksi kyseinen luonnollisella kielellä kirjoitettu aineisto on kuitenkin muunnettava ohjelmallisesti analysoitavaan muotoon irrottamalla lähdetekstistä merkityksellinen data jatkokäsittelyä varten.

Tätä tehtävää varten *Kairan* ensimmäinen prototyyppi kehitettiin vuonna 2015 Lammin Biologisella asemalla. Vuonna 2017 prototyypin digitoiman datan pohjalta käynnistyi Helsingin Yliopiston tutkimushanke *Learning from our past: the effect of forced migration from Karelia on family life* [18], jonka osana *Kairan Siirtokarjalaisten tie* -kirjasarjasta digitoimasta datasta rakennettiin tietokanta tieteellistä tutkimusta varten.

Tutkimusprojektin jatkuessa sovellukseen tullaan lisäämään uusia ominaisuuksia, ja sen irrottamaa dataa laajennetaan. Myöhemmin sovellusta voidaan soveltaa myös muihin *Siirtokarjalaisten tie* -kirjasarjaa vastaaviin matrikkeleihin. Nämä vaatimukset puolsivat sovelluskehysarkkitehtuurin kehittämistä, joka tarjoaisi selkeän arkkitehtuurin erilaisten tiedon louhinta- ja käsittelyalgoritmien yhteensovittamiselle kirjasarjakohtaisiksi kokonaisuuksiksi.

Tässä työssä suunnitellaan ja toteutetaan *Kairan* yleinen arkkitehtuuri ja sovelluksen rakenne. Tiedon louhintaan liittyvät algoritmit ja menetelmät on rajattu tämän työn ulkopuolelle esimerkkitapauksia lukuunottamatta. Luvussa 2 käydään lyhyesti läpi sovellusarkkitehtuurien peruskäsitteet sekä menetelmiä sovellusarkkitehtuurien arvioimiseksi. 3 luvussa kuvataan tarkemmin tietovuoarkkitehtuurin teoria, joka valittiin sovelluskehyksen toteutuksen ensisijaiseksi arkkitehtuuriksi. Luvussa 4 kuvataan sovelluksen toteutus pääpiirteissään ja kuvataan sen tärkeimmät ominaisuudet. Luvussa 5 arvioidaan toteutusta luvussa 2 esiteltyjen arviointimenetelmien mukaisesti sekä arvioidaan potentiaalisia jatkokehityskohteita.



KUUTIO, AHTO
maanvilj., synt. 11. 9. -08 Metsäpirtillä. Puol. Essi Esimerkki emäntä, synt. 23. 5. -13 Käkisalmella. Lapset: Lasse Lapsi -46, Lassi Lapsi -49. Molemmat syntyneet Seinäjoella. Asuinp. Karjalassa: Metsäpirtti -39. Muut asuinp.: Juva, Kangasniemi, Heinävesi, Seinäjoen mlk. 41—50, Lieto 50—. Kuutiot ovat saaneet nykyisen tilansa maanlunastuslautakunnan kautta ja ovat itse rakentaneet kaikki rakennukset. Tilan pinta-ala on 11 ha. Ahto Kuutio on palvellut korpraalina talvisodassa Metsäpirtin RajaK:ssa ja jatkosodassa PionK:ssa. Hän on toiminut Metsämäen laidunyhtymän puheenjohtajana n. 10 vuotta.

***Kuva 1**. Anonymisoitu ote kirjasarjan henkilöstä. Tekstissä kuvaillaan lyhyesti henkilöiden elämänvaiheita ja varallisuutta.*

```
1 <DATA bookseries="siirtokarjalaiset" book_number="2">
2   <PERSON name="KUUTIO,␣AHTO" approximated_page="507-509">
3     <RAW>
4       maanvilj., synt. 11. 9. -08 Metsäpirtillä. Puol. Essi Esimerkki,
5       emäntä, synt. 23. 5.-13 Käkisalmella. Lapset: Lasse Lapsi 46,
6       Lassi Lapsi -49.
7     </RAW>
8   </PERSON>
9 </DATA>
```

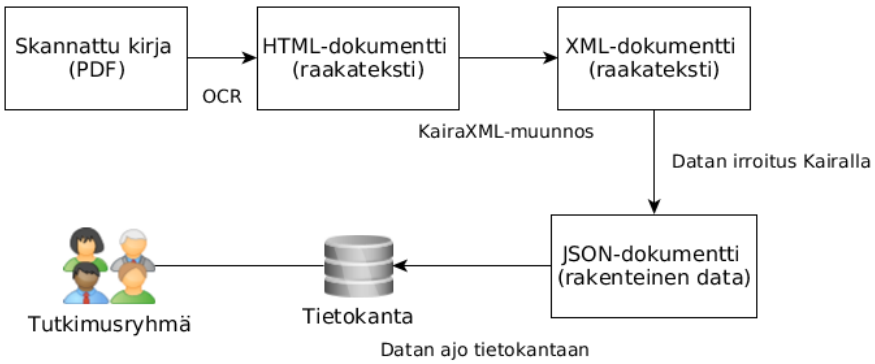
***Listaus 1**. Lyhennetty näyte XML-dokumentin formaatista.*

Alkuperäiset matrikkelikirjat julkaistiin 1970-luvulla, joten niistä ei ole olemassa virallisia sähköisiä painoksia, joiden teksti olisi digitoitu koneluettavaan muotoon. Kirjasarjoja on kuitenkin skannattu paperiformaatista PDF-formaattiin, mikä mahdollistaa digitaalisen tekstin tuottamisen niistä tekstintunnistussovellusten (OCR, *Optical Character Recognition*) avulla.

Tutkimusprojektissa tekstintunnistukseen käytettiin kaupallista tekstintunnistussovellusta ABBYY Finereader [1], jolla kirjasarjat muunnettiin HTML-dokumenteiksi säilyttäen pääpiirteittäin alkuperäisen kirjasarjan ladonnan ja kappalejaon. Kirjasarjoissa jokaiselle henkilölle on jaettu selkeä kappale (kuva 1), joka voidaan tunnistaa HTML-dokumentista ohjelmallisesti.

Tekstintunnistusohjelman tuottamasta HTML-dokumentista tuotettiin jäsentämällä yksinkertaistettu XML-formaatti, jossa kukin dokumentin elementti sisältää yhden kirjasarjan henkilöä käsittelevän kappaleen raakatekstin. Useimmat tekstikappaleet sisältävät kuitenkin kuvauksen kahdesta luonnollisesta henkilöstä: ensisijaisesta henkilöstä sekä tämän mahdollisesta puolisosta. Yleensä pariskunnan lapset on myös listattu lyhyesti syntymävuosineen. XML-formaattia on havainnollistettu listauksessa 1.

XML-dokumentti voidaan sitten syöttää *Kairaan*, joka irrottaa tekstistä tutkimuksen kannalta kiinnostavan informaation ja tuottaa ulostulona JSON-dokumentin.



Kuva 2. *Datan polku fyysisistä kirjoista tutkimusryhmän käyttöön. Kaira muuntaa yksinkertaisen XML-raakatekstin rakenteiseksi JSON-formaatiksi.*

Tämä rakenteinen dokumentti voidaan sitten ajaa relaatiotietokantaan myöhempää tutkimustyötä varten. Tämä datan polku alkuperäisistä fyysisistä kirjoista tutkimusryhmän tilastollista analyysiä varten on havainnollistettu kuvassa 2.

Sovelluskehys voi käyttää syötteenään myös muita dataformaatteja, esimerkiksi luonnollisen kielen lauserakenteiden annotointiin käytettävää *CoNLL-U* formaattia [5]. Tämä kuitenkin edellyttää, että kirjasarjan analysointiin käytettävät ohjelmistokomponentit pystyvät tulkitsemaan syötedataa.

Diplomityö on soveltava työ, jonka käsittelemä sovellus on rakennettu tutkimusprojektin tavoitteiden mukaisiksi. Siten tässä työssä esitellyt metodologiat ja niiden perustelut juontuvat pitkälti tutkimusprojektin tavoitteista ja vaatimuksista.

2. OHJELMISTOARKKITEHTUURIEN SUUNNITTELU JA NIIDEN ARVIOINTI

Ohjelmistoarkkitehtuurilla on useita erilaisia määritelmiä. Yhden määritelmän mukaan ohjelmistoarkkitehtuuri on korkean tason kuvaus tietokonesovelluksen rakenteesta, sen osista ja näiden osien suhteista toisiinsa [3]. Sovellusten ohjelmistoarkkitehtuurien arviointiin on kehitetty lukuisia menetelmiä, jotka keskittyvät sovellusarkkitehtuurien eri ominaisuuksiin. Esimerkiksi teollisuudessa hyödynnettävä ATAM (*architecture tradeoff analysis method*) arvioi kuinka hyvin arkkitehtuuri vastaa sille asetettuihin laatuvaatimuksiin ja näiden laatuvaatimusten välisiin suhteisiin. ATAM ja lukuisat muut arviointimenetelmät perustuvat skenaarioihin, jotka ilmentävät järjestelmän kykyä vastata suunniteltuihin laatuvaatimuksiin. [20]

Kaikki arviointimenetelmät eivät kuitenkaan perustu skenaarioihin. Esimerkiksi DCAR (*decision-centric architecture review method*) on menetelmä joka kiinnittää huomiota erityisesti arkkitehtuuripäätösten takana oleviin syihin. Skenaarioiden sijasta menetelmässä tarkastellaan arkkitehtuuriin vaikuttavia tekijöitä, jotka rajoittavat ja määrittelevät suunnittelupäätöksiä. DCARin tavoitteena on myös keventää arkkitehtuurien arviointiprosessia, jotta sitä voitaisiin harjoittaa iteratiivisesti ohjelmistoprojektin aikana [8]. Muita mahdollisia menetelmiä ohjelmistoarkkitehtuurien arviointiin ovat esimerkiksi sovelluskehittäjien domain-tuntemukseen perustuva kokemuspohjainen arviointi, matemaattinen mallintaminen ja simulaatiot [20].

Tässä luvussa käsitellään *Kairan* kannalta oleellisia ohjelmistoarkkitehtuurien arviointiin liittyviä menetelmiä, joita on sovellettu ohjelmistoprojekteissa. Perinteisesti ketterässä ohjelmistokehityksessä muodolliset arkkitehtuurien suunnittelumenetelmät on nähty raskaina tai kalliina ja parhaiden arkkitehtuurien on nähty syntyvän tiimiltä itseltään. [8][4] Toisaalta korkean tason arkkitehtuurikuvaukset on koettu hyödyllisiksi erityisesti suurissa tietojärjestelmissä, sillä yleensä niiden ylläpidettävyys, luotettavuus ja jatkokehityksen joustavuus ovat riippuvaisia tehdyistä arkkitehtuurivalinnoista ja niistä seuraavista kehitysperiaatteista. Täten ohjelmistoarkkitehtuurien dokumentointia ja arviointia pidetään kuitenkin tärkeänä osana onnistunutta ohjelmistoprojektia [8]. Tästä syystä arkkitehtuurien suunnittelu- ja arviointimenetelmät koettiin tarpeellisiksi *Kairan* arkkitehtuurin arvioinnille. Luvussa 5 kuvattuja arkkitehtuurien arviointimenetelmiä sovelletaan käytännössä *Kairan* arkkitehtuurin arviointiin.

2.1 Skenaariot lähtökohtana arkkitehtuurin suunnittelussa

Skenaariot ovat tapa analysoida sovelluksen laatuvaatimusten toteutumista. Järjestelmän arkkitehtuurin on tavallisesti toteutettava varsinaisten toiminnallisten vaatimusten lisäksi myös laadullisia ominaisuuksia. Tällaisia laadullisia ominaisuuksia voivat olla esimerkiksi järjestelmän laajennettavuus, ylläpidettävyys, testattavuus ja käytettävyys. Vaikka nämä laatuvaatimukset eivät välttämättä vastaa toiminnallisia vaatimuksia, on ne huomioitava arkkitehtuuria suunniteltaessa. [3, luku 4]

Järjestelmän laajennettavuus ei itsessään tarkoita varsinaisesti mitään, sillä periaatteessa kaikki järjestelmät ovat jossain määrin muokattavissa ja laajennettavissa. Laajennettavuus voidaankin nähdä laadullisena ominaisuutena. Kuinka nopeaa järjestelmää on laajentaa? Millaisia muutoksia lähdekoodiin on mahdollista tehdä ilman mittavaa uudelleenkirjoitustyötä? Laadulliset ominaisuudet ovat siten tärkeä osa ohjelmistoarkkitehtuurin suunnittelutavoitteita. Niiden tunnistaminen ja niiden toteutumisen arviointi lopullisessa ohjelmistoarkkitehtuurissa on tärkeää[3, luku 4].

Skenaariomenetelmissä luodaan skenaarioita, jotka pyrkivät mahdollisimman konkreettisesti kuvailemaan sovellukselle asetetun laatuvaatimuksen. Näiden kuvausten perusteella arkkitehtuuria voidaan tarkastella huomioiden erityisesti millaisia seurauksia arkkitehtuurista kussakin skenaariossa syntyy [19]. Havaittujen seurausten pohjalta voidaan arvioida vastaako sovellus todella suunniteltuihin vaatimuksiin ja tarvitseeko sitä kehittää toimimaan paremmin tietyissä skenaarioissa.

Skenaarioihin pohjautuvilla arkkitehtuurien arviointimenetelmillä on erilaisia tavoitteita ja lähestymistapoja. Aiemmin kuvaillun ATAMin lisäksi esimerkiksi ALMA *(architecture level modifiability analysis)* pyrkii ennustamaan ohjelmistoprojektin ylläpidettävyyden kustannuksia ja riskien arviointia. Menetelmiä voidaan soveltaa projektin eri vaiheissa vaihdellen arkkitehtuurin suunnitteluvaiheesta sen jälkikäteiseen arviointiin [20]. Luonnollisesti arvioinnin ajankohta projektin elinkaaren aikana vaikuttaa arvioinnissa syntyneiden johtopäätösten sovellettavuuteen. Varhaisessa vaiheessa tehty arviointi muuttaa sovelluksen arkkitehtuuria ennen sen toteutusta. Projektin loppuvaiheessa tehty arkkitehtuurin arviointi taas voi osoittautua hyödylliseksi muiden myöhempien sovellusten suunnittelussa ja toteutuksessa.

2.1.1 Laadullisten ominaisuuksien tunnistaminen

Arkkitehtuurin tarvitsemia laadullisia ominaisuuksia voidaan yrittää tunnistaa kehittämällä skenaarioita, jotka vaativat tiettyjen laatuvaatimusten toteutumista. Skenaariot pyritään luomaan konkreettisiksi arkkitehtuurin suunnittelun kannalta. Esimerkiksi järjestelmän muokattavuudesta voitaisiin esittää seuraavanlainen hypoteettinen skenaario:

"Järjestelmään on lisättävä komponentti, joka mahdollistaa tulosteen tallentamisen uuteen tiedostoformaattiin. Uuden komponentin lisääminen ja testaaminen on tehtävissä muuttamatta muita järjestelmän komponentteja."

Tämä skenaario ilmentää laatuvaatimusta, joka vaikuttaa sovelluksen konkreettiseen arkkitehtuuriin. Vaatimuksen toteutumiseksi arkkitehtuurin on esimerkiksi tuettava toiminnallisuuden rajaamista erillisiin moduuleihin ja määriteltävä yhteiset rajapinnat komponenttien välille.

Skenaarioiden tuottamiseen voidaan käyttää erilaisia malleja ja menetelmiä. Felix Bachmann ja Mark Klein[3, luku 4] ovat esitelleet mallin, jossa laadullisen ominaisuuden havainnollistava skenaario on jaettu kuuteen osaan. Skenaarion osat kuvaavat esimerkiksi:

- Stimulin lähde eli skenaarion alkuunpanija tai toimija.
- Stimuli on tapahtuma johon järjestelmän on jotenkin reagoitava.
- Ympäristö, jossa stimuli tapahtuu.
- Artifakti on kohde, johon stimuli vaikuttaa järjestelmässä.
- Vastaus on toimenpide, joka stimulista seuraa.
- Vastauksen mitta on menetelmä, jolla stimulin aiheuttamaa vastausta voidaan testata tai mitata.

Tämän mallin pohjalta voidaan kehittää kaavanmukaisia skenaarioita, joiden pohjalta arkkitehtuurille voidaan asettaa laadullisia vaatimuksia. Syntyneiden skenaarioiden pohjalta voidaan tunnistaa mahdollisesti aikaisemmin huomioimatta jääneet laatuvaatimukset, jotka tulisi huomioida sovelluksen arkkitehtuurissa.

Arkkitehtuuri voidaankin suunnitella rekursiivisesti komponenttitaso kerrallaan käyttäen lähtökohtana joukkoa laatuvaatimuksia ja funktionaalisia vaatimuksia, jotka on tunnistettu skenaarioiden pohjalta. Tätä menetelmää kutsutaan *attribuuttivetoiseksi suunnitteluksi (ADD - attribute driven design)*[3, luku 7]. Menetelmässä arkkitehtuuri jaetaan osiin, joista kullekin hahmotellaan rakenne perustuen niihin liittyviin vaatimuksiin. Rakenteessa otetaan huomioon järjestelmän osien suhteet toisiinsa sekä mahdolliset suunnittelumallit, joilla rakenne voidaan toteuttaa. Teknisten yksityiskohtien suunnittelu jätetään kuitenkin myöhempään ajankohtaan.

2.1.2 Taktiikat laatuominaisuuksien saavuttamiseen ja arkkitehtuurin suunnittelun avuksi

Laadullisten ominaisuuksien tunnistaminen ei itsessään riitä niiden saavuttamiseen. Tähän tarvitaan *taktiikoita* eli suunnittelupäätöksiä, jotka auttavat tai mahdollista-

vat laatuvaatimuksen täyttämisen. Aiemmin esitellyssä esimerkkiskenaariossa sovelluksen laajennettavuuden toteuttaminen vaatii suunnittelupäätöksiä, jotka tukevat havaittua laadullista vaatimusta. [3, luku 5]

Muunneltavuustaktiikoiden tavoitteena on tukea ohjelmiston muunneltavuuteen liittyviä laadullisia vaatimuksia. Yksi esimerkki tällaisesta taktiikasta on *muutosten lokalisointi*. Tässä arkkitehtuuritaktiikassa pyritään rajaamaan sovelluksen toiminnallisuus moduuleihin, joille on annettu niille ominaiset roolit. Tällöin mahdolliset muutokset toiminnallisuuteen rajoittuvat tietyn moduulin sisäiseen toteutukseen vaatimatta muutoksia muuhun sovellukseen. Niin sanottu yhden vastuun suunnitteluperiaate on toinen ilmaisu tälle taktiikalle, jonka mukaan kullakin ohjelmistokomponentilla tulisi olla sille tarkkaan määritelty yksittäinen vastuualue.

Taulukossa 2 on listattu Clements et al. kuvailemia muunneltavuustaktiikoita ja lyhyet kuvaukset niiden vaikutuksista arkkitehtuuriin[3, luku 5].

Taulukko 2. *Yleisiä muunneltavuustaktiikoita ja laatuvaatimukset, jotka ne pyrkivät toteuttamaan*

Taktiikka	**Kuvaus**
Muutosten lokalisointi	Moduuleille asetetaan tarkkaan määritellyt roolit, jolloin niihin tehtävät muutokset rajoittuvat pieneen osaan koodikantaa. Muokkausten rajoittuminen täsmälliseen osaan sovellusta vähentää muutosten kustannuksia.
Semanttinen koherenssi	Moduulit sisältävät komponentteja, joilla on ensisijaisesti suhteita toisiinsa. Ne toimivat yhdessä ilman liiallista tarvetta ulkoisille moduuleille. Tämä rajaa mahdolliset muutokset moduulin sisäisiin komponentteihin ja niiden vastuualueisiin.
Moduulin yleistäminen	Moduuli yleistetään siten, että se kykenee suorittamaan laaja-alaisempia tehtäviä riippuen sen parametrisoinnista. Tällöin mahdolliset muutokset voidaan tehdä muokkaamalla moduulille välitettäviä parametreja.
Tulevaisuuden muutosten odottaminen	Järjestelmän moduulien vastuita suunniteltaessa harkitaan rajoittavatko tehdyt päätökset tarvittavien muutoksien laajuutta, kun järjestelmää joudutaan muokkaamaan odotetulla tavalla. Odotettavissa olevien muutosten vaikutukset pyritään minimoimaan huomioimalla ne jo aikaisemmassa suunnittelutyössä.

Muunneltavuustaktiikoihin liittyvät myös värevaikutukset *(ripple effects)*, jolloin koodikantaan tehty muutos väreilee muualle koodikantaan edellyttäen muutoksia

myös muihin sovelluksen osiin. Nämä tarvittavat muutokset voivat vastaavasti väreillä eteenpäin ja edellyttää lisää muutoksia. Näin muutos yhteen paikkaan koodikantaa voi käynnistää suuren ja työlään muutosketjun. Värevaikutuksia voivat aiheuttaa useat erilaiset riippuvuustavat, joilla sovelluksen eri moduulit ovat riippuvaisia toisistaan. Suoran toisen moduulin palveluiden käyttämisen lisäksi riippuvuussuhteita voi syntyä muun muassa rajapintojen, odotetun dataformaatin tai suoritusjärjestyksen kautta.

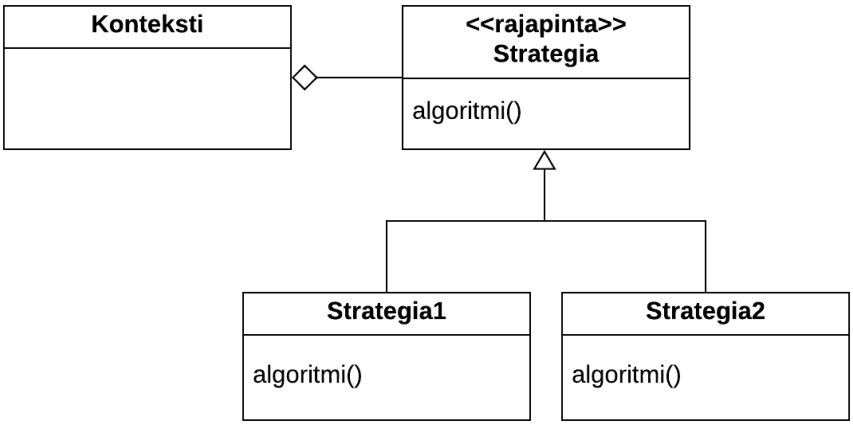
Clements et al. ovat listanneet värevaikutusten minimoimiseksi taktiikoita[3, luku 5], joiden nimet ja perusideat on listattu taulukkoon 3.

Taulukko 3. *Taktiikoita värevaikutusten minimointiin arkkitehtuurissa*

Taktiikka	**Kuvaus**
Tiedon piilottaminen	Moduulin sisäiset vastuut jaetaan julkisiin ja yksityisiin vastuisiin. Julkiset vastuut paljastetaan moduulin ulkopuolelle rajapinnan kautta, jolloin moduulin sisäisen toteutuksen muutokset eristyvät moduulin sisäpuolelle.
Olemassa olevien rajapintojen ylläpito	Mikäli mahdollista, olemassa olevia rajapintoja ei tulisi muuttaa moduulien välillä. Tällöin muutostarve ei välity muihin rajapintaa käyttäviin moduuleihin. Huomautettakoon kuitenkin, ettei tämä välttämättä ole mahdollista, mikäli datan tai palveluiden merkitykset muuttuvat.
Kommunikaatioreittien rajoittaminen	Rajoita moduulien määrää jotka ovat riippuvaisia moduulista ja sen tuottamasta datasta. Vastaavasti rajoita moduulien määrää, joista moduuli itse on riippuvainen. Tällöin mahdolliset muutokset moduuleihin välittyvät vain rajalliseen määrään muita moduuleita.
Välikappaleet	Vaihdellen riippuvuustyypistä on mahdollista lisätä moduulien riippuvuussuhteeseen välikappale, joka käsittelee riippuvuussuhteeseen liittyviä operaatioita. Välikappale voi esimerkiksi muuntaa datan formaattia tarvittaessa tai tarjota yksinkertaistetun rajapinnan riippuvuuden kohteena olevan moduulin käyttöön (esim. facade-suunnittelumalli [15]). Välikappale katkaisee riippuvuusketjun ja rajaa muutostarpeiden välittymistä ketjussa eteenpäin.

Kun tarve taktiikalle on havaittu, voi arkkitehti pohtia sopivaa suunnittelumallia, jolla toteuttaa taktiikka ohjelmakoodissa. Esimerkissämme tämä voisi tarkoittaa olio-ohjelmoinnin *strategia-suunnittelumallin* käyttöä.

Strategia-suunnittelumallissa ohjelmistokomponentin toteutus piilotetaan yleisem-



***Kuva 3**. Strategia-suunnittelumallissa useat toisistaan poikkevat luokat toteuttavat yhteisen rajapinnan.*

män rajapinnan alle. Tarvittaessa toiminnallisuutta voidaan muokata vaihtamalla luokka toiseen luokkaan, joka toteuttaa saman määritellyn rajapinnan, mutta jonka sisäinen toiminta on erilainen. Kuvassa 3 on kuvattu strategialle tyypillinen luokkakaavio. Useampi luokka toteuttaa saman strategian rajapinnan, jolloin ulompi konteksti voi kutsua mitä tahansa rajapinnan toteuttavaa luokkaa. Jokainen rajapinnan toteuttava luokka toteuttaa erilaisen algoritmin. [15]

Strategiaa voitaisiin käyttää esimerkiksi erilaisten tiedostojen lukuoperaatioiden toteuttamiseen, joissa jokaisella eri tyyppisellä tiedostolla on erilainen datan luku- ja käsittelyalgoritminsa [15]. Strategia on suunnittelumalli, jonka voi katsoa toteuttavan siten taktiikat tiedon piilottamiselle (algoritmien kapselointi erillisiin luokkiin), rajapintojen säilyttämiselle (uudet algoritmit toteuttavat saman rajapinnan), tulevaisuuden muutosten odottamiselle (tarjoaa variaatiopisteen uuden algoritmin lisäämiselle) ja muutosten lokalisoinnille (kunkin algoritmin toteutus omassa luokassaan).

Toisaalta strategia suunnittelumallina rajoittuu tarjoamaan erilaisia toimintamalleja samalle käyttäytymiselle, joten malli toimii vain tällaisissa käyttötapauksissa. Rajapintamuutoksien ilmaantuessa on kaikkien strategian toteuttavien luokkien rajapintoja muutettava. Ohjelmistoarkkitehdin onkin tehtävä valintoja käytettävistä suunnittelumalleista ja pohdittava mitä taktiikoita nämä mallit edustavat ja millaisia vaikutuksia valituilla malleilla on moduulien välisiin suhteisiin.

Suunnittelumalleista puhuttaessa on kuitenkin huomautettava, että kaikki toteutusmenetelmät eivät itsessään ole suunnittelumalleja, vaikka ne olisivatkin onnistuneita ratkaisuja sovelluksen arkkitehtuurissa ilmenevään ongelmaan ja toteuttavat arkkitehtuurissa tarvittavat taktiikat. Suunnittelumallin määritelmän mukaan niitä on täytynyt soveltaa onnistuneesti useita kertoja eri yhteyksissä. Mikäli käsillä olevaan

arkkitehtuuriongelmaan ei ole olemassa valmista suunnittelumallia, on arkkitehdin kehitettävä oma ratkaisunsa, jolla toteuttaa tarvittavat taktiikat ja siten laatuvaatimukset sovelluksessaan [21].

2.2 Refaktorointi arkkitehtuurisuunnittelun osana

Pitkäkestoisissa ohjelmistoprojekteissa ohjelmakoodin laatu korostuu. Koodia luetaan huomattavasti useammin kuin sitä kirjoitetaan, sillä yleensä muutokset ohjelmakoodiin vaativat olemassa olevan koodin toiminnallisuuden ymmärtämistä. Tämä tarkoittaa, että koodia on kehityksen aikana muokattava selkeämmäksi ja rakenteeltaan paremmaksi ilman toiminnallisuuden muuttumista. Tätä prosessia kutsutaan refaktoroinniksi. [14]

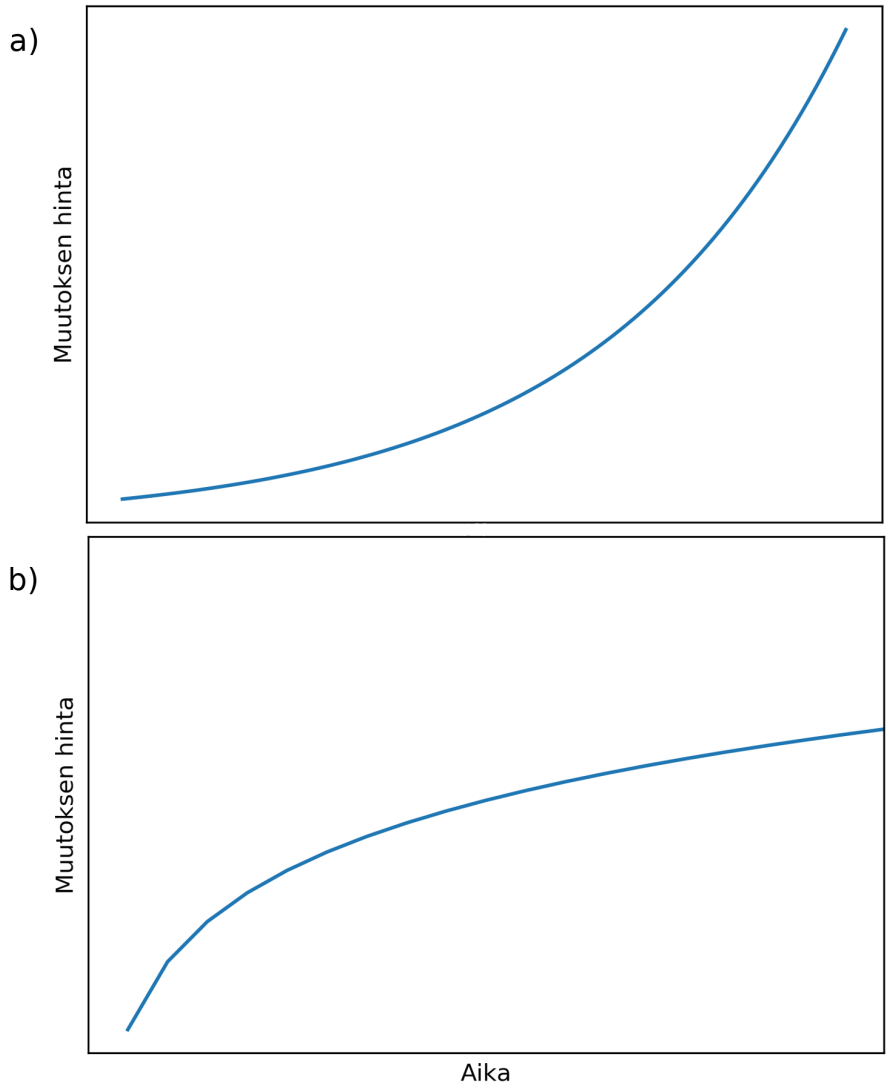
Koska refaktoroidessa olemassa olevaa koodia muokataan rakenteeltaan tarkoituksena olla muokkaamatta sen toiminnallisuutta, vaaditaan prosessiin yleensä kattavat automaatiotestit. Automaatiotestit tarjoavat muuten riskialttiille prosessille turvaverkon, jolla voidaan vähentää refaktoroinnista aiheutuvien ohjelmistovirheiden syntyä. Täten esimerkiksi Martin Fowler pitää automaattitestausta refaktoroinnin mahdollistavana tekijänä ja välttämättömänä osana käytäntöä [12].

Refaktorointi voi vaihdella yksinkertaisesta yksittäisten funktioiden siistimisestä ja jäsentelystä laajamittaisempiin arkkitehtuuriin vaikuttaviin muutoksiin. Jälkimmäinen refaktorointityyppi onkin tärkeä osa arkkitehtuurisuunnittelua ja sen ylläpitoa. Esimerkiksi ohjelmistokehysten lopullinen muoto harvoin syntyy kerralla ja vaatii siten refaktorointia lopullisen arkkitehtuurin löytämiseksi [14].

Evolutiivinen arkkitehtuurisuunnittelu

Evolutiivisessa arkkitehtuurisuunnittelussa järjestelmän rakenne kasvaa toteutuksen edetessä. Huonosti toteutettuna tämä periaate voi johtaa *adhoc*-ratkaisuihin ja kokonaisuudeltaan muodottomaan arkkitehtuuriin, joka kasvaa ohjelmistoprojektin edetessä vaikeasti ylläpidettäväksi ja laajennettavaksi. Tätä projektiin tehtävien muutosten vaikeutumista voidaan kuvata *muutoskäyrällä*. Muutoskäyrä kuvaa ohjelmistoon tehtävien muutosten hintaa projektin edetessä ja sen ajatellaan usein olevan eksponentiaalisesti kasvava (kuva 4, kuvaaja a). Tämän perusteella voitaisiin päätellä ettei evolutiivinen arkkitehtuurisuunnittelu voi toimia, sillä jälkikäteiset muutokset olisivat hyvin kalliita toteuttaa. [12]

Fowler kuitenkin esittää menetelmiä, joilla muutoskäyrää voidaan litistää (kuva 4, kuvaaja b) ja siten tehdä myöhäisetkin muutokset arkkitehtuuriin kustannustehok-



Kuva 4. *Havainnekuvat muutoskäyristä. Kuvaaja a) on perinteinen muutoskäyrä, jossa ohjelmistoprojektin muutosten hinta kasvaa ajan myötä voimakkaasti. Kuvaaja b) on litistetty muutoskäyrä.*

kaiksi mahdollistaen evolutiivisen arkkitehtuurisuunnittelun ja esimerkiksi suunnittelupäätösten viivästämisen. Menetelmistä tärkeimpiä ovat ohjelmistotestaus, jatkuva integraatio ja refaktorointi. Ennakoivalla arkkitehtuurisuunnittelulla on edelleen tärkeä rooli sovelluksen onnistuneessa suunnittelussa, mutta etenkin regressiotestaus ja sen mahdollistama refaktorointi mahdollistavat sovelluksen suunnittelupäätösten muokkaamisen projektin myöhemmässä vaiheessa. [12] Myöhempää laajennettavuutta ja muokattavuutta helpottavat myös olemassa olevan arkkitehtuurin toteuttamat laajennettavuutta tukevat taktiikat.

Käytännössä tässä evolutiivisessa arkkitehtuurisuunnittelussa arkkitehti punnitsee tehtävien arkkitehtuuripäätösten ajankohtaa. Ajoittain suunnittelupäätös kannattaa viivästää myöhempään ajankohtaan, kun domain-tuntemusta on enemmän saatavilla. Näin voidaan esimerkiksi välttää turhien ominaisuuksien toteuttaminen ennakkoon säästäen siten aikaa ja rahaa.

Suunnittelupäätösten viivästämistä havainnollistetaan joskus YAGNI-periaatteella *(You Ain't Gonna Need It)*, jonka mukaan sovellukseen ei tulisi lisätä koodia, jota käytetään vasta myöhempien ominaisuuksien toteutuksessa. Tämä periaate vaatii kuitenkin harkintaa sovelluskehysten ja uudelleenkäytettävien komponenttien kirjoittamisen yhteydessä, sillä ne ovat monimutkaisia ohjelmistoyksiköitä, joiden tavoitteena on alkuinvestoinnin jälkeen helpottaa jatkokehitystä. Toisaalta liian varhain rakennetut komponentit saattavat toteuttaa tehtävänsä huonosti asiakasvaatimusten kehittyessä tai alkuvaiheen vähäisen domain-tuntemuksen vuoksi. Väärään tai huonoon ratkaisuun käytetty aika tarkoittaa hukkaan heitettyä aikaa ja rahaa.

Toisaalta koska refaktoroinnista on kattavalla testauksella tehty mahdollista, voidaan mahdolliset huonot suunnittelupäätökset korjata suhteellisen vaivattomasti jälkeenpäin. Lisäksi aiemmin YAGNI-periaatteella kehitetyt yksinkertaiset arkkitehtuuriratkaisut voidaan muokata uusia ominaisuuksia tukeviksi monimutkaisemmiksi ja joustavammiksi rakenteiksi. Tämä mahdollistaa järjestelmän arkkitehtuurin evolutiivisen kehittämisen ja mahdollistaa vastaamisen muuttuviin asiakasvaatimuksiin samalla minimoiden ennenaikaisen kehitystyön ja sen sisältämät riskit. [12]

3. SOVELLUSKEHYKSET JA TIETOVUOARKKITEHTUURIT

Tässä luvussa käydään läpi lyhyesti *Kairan* toteutukselle tärkeimmät sovellusarkkitehtuurin käsitteet ja teoria, jotka auttavat lukijaa ymmärtämään luvussa 4 kuvattua sovelluksen toteutusta ja siinä tehtyjä valintoja. *Kairan* kokonaisuutta voidaan kutsua tietovuoarkkitehtuurityylillä rakennetuksi sovelluskehykseksi.

3.1 Sovelluskehykset ja niiden erikoistaminen

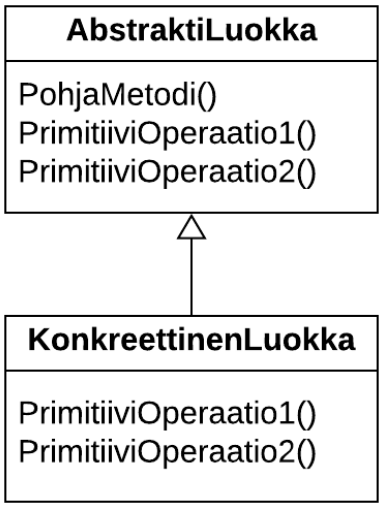
Sovelluskehykset tarjoavat kokoelman luokkia tai muita ohjelmistomoduuleita ja rajapintoja. Sovelluskehysten tarjoama perustoiminnallisuus erikoistetaan lopulliseksi tuotteeksi laajentamalla sovelluskohtaista toiminnallisuutta kehyksen toiminnallisuuden ympärille. Näin voidaan kehittää useita saman ydintoiminnallisuuden jakavia erikoistettuja sovelluksia, jotka muodostavat yhdessä sovellusperheen. Sovellusperhe on samankaltaisten sovellusten joukko, joilla on yhteistä toiminnallisuutta. Sovelluskehykset ovatkin tapa toteuttaa *tuoterunkoarkkitehtuuri*, jonka tehtävänä on helpottaa sovellusperheiden laajentamista ja ylläpitoa systemaattisen koodin uudelleenkäytön avulla [17].

3.1.1 Muunneltavuus

Uudet tuoteperheen sovellukset kehitetään muuntelemalla ja erikoistamalla ne tuoterunkoarkkitehtuurista. Sovelluskehyksien tapauksessa tämä yleensä tapahtuu sovelluskehyksen laajennosrajapinnan kautta. Sovelluskehyksen runkoon täydennetään lopullisen sovelluksen tarvitsema toiminnallisuus kokonaisarkkitehtuurin pysyessä edelleen sovelluskehyksen määrittelemien suuntaviivojen mukaisina.

Sovelluskehyksiä suunniteltaessa suunnittelijan on tehtävä päätöksiä kehyksen rajapinnasta ja kehyksen puolelle sisällytettävästä toiminnallisuudesta, jotta syntyvä kehys vastaisi tarvittaviin muunneltavuusvaatimuksiin. Toisaalta kehyksen tulisi tarjota vakaa yhteinen rakenne kaikille siitä erikoistettaville sovellusperheen jäsenille. Sovelluskehyksen on vastattava sovellusperheen yhteisiin laatuvaatimuksiin tai ainakin tuettava niiden saavuttamista.

Piirremallit ovat yksi keino hahmottaa sovellusten muunteluvaatimuksia. Piirrekartoituksessa suunnittelija pyrkii tunnistamaan kaikille sovellusperheen osille yhteisen



***Kuva 5**. Hollywood-periaatteessa luokat erikoistetaan abstraktista luokasta. Abstrakti luokka määrittelee pohjametodin, joka kutsuu erikoistettujen luokkien metodeita suorittaen niiden toteutuksen.*

toiminnallisuuden sekä vastaavasti sovellusten toiminnalliset erot. Mallissa voidaan havainnollistaa sovelluksen pakolliset ominaisuudet, vaihtoehtoiset ominaisuudet sekä näiden piirteiden välisiä rajoitteita. Mallit voidaan ilmaista laajennetulla UML-notaation mukaisena luokkakaaviona [17].

3.1.2 Erikoistamisrajapinta ja erikoistamismekanismit

Kun tärkeimmät muunneltavuusvaatimukset on tunnistettu ja sovelluskehyksen vastuu tuoteperheen toiminnallisuuden toteutuksesta on kartoitettu, voidaan alkaa suunnittelemaan sovelluskehyksen muunneltavuuden toteuttavaa rajapintaa. Rajapinnan kautta toteutetaan lopulliset yksittäiset sovellukset.

Erikoistamisrajapinnan toteutuksessa voidaan hyödyntää suunnittelumalleja, joilla voidaan lisätä sovelluskehyksen tarvitsemaa joustavuutta [17]. Suunnittelumallit kuvaavat ohjelmistokehityksessä usein ilmeneviä suunnitteluongelmia ja näihin hyviksi havaittuja ratkaisuja yleisessä muodossa [15].

Sovelluskehyksen rajapinnassa voidaan hyödyntää *Hollywood-periaatetta* [13], jossa sovelluksen suoritusvastuu on kehyksellä, joka kutsuu rajapintansa kautta erikoistettua toiminnallisuutta. Näin erikoistettava toiminnallisuus vain liitetään osaksi sovelluskehyksen muodostamaa kokonaisuutta, ja kontrolli ohjelman suorituksesta pysyy pääosin kehyksellä. Tämä mahdollistaa sovelluksen erikoistamisen kehyksen määrittelemän erikoistamisrajapinnan kautta ilman, että sovelluskohtaisten komponenttien tarvitsee huolehtia koko sovelluksen suoritusjärjestyksestä ja siihen liittyvistä

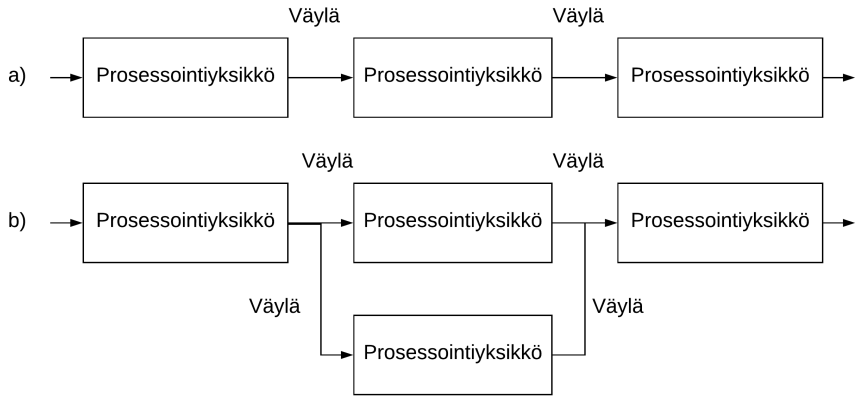
vastuista. Kuvassa 5 on esitetty mahdollinen luokkakaavio Hollywood-periaatteen toteuttamiseksi. Erikoistettu toiminnallisuus on eriytetty luokkiin, jotka perivät abstraktin kantaluokan. Sovelluskehys kutsuu kantaluokan *pohjametodia*, joka puolestaan kutsuu erikoistettujen luokkien primitiivioperaatioita. Kantaluokan tehtävä on määrittää erikoistetuille luokille yhtenevä algoritmin rakenne. Näin algoritmin suoritusvastuu pysyy sovelluskehyksen hallinnassa, sillä kantaluokan rakenne on sovelluskehyksen määrittelemä. [15]

Sovelluskehykset voidaan karkeasti jaotella kolmeen kategoriaan niiden erikoistamismekanismien mukaan. *Muunneltavissa kehyksissä* erikoistaminen tapahtuu sovelluskehyksen tarjoamien kantaluokkien avulla, joista periyttämällä luodaan sovelluskohtaisia aliluokkia. Aliluokkien toteutus ja kantaluokan toiminnallisuuden laajentaminen mahdollistavat suhteellisen vapaan kehyksen erikoistamisen, edellyttäen että sovelluskehittäjä tuntee sovelluskehyksen rajapinnan toteutuksen, vaatimukset ja oletukset riittävän hyvin.

Rajoittuneemman tavan erikoistamiseen tarjoavat *koottavat kehykset*, joissa erikoistaminen toteutetaan ensisijaisesti kehyksen tarjoamien valmiiden komponenttien parametrisoinnilla. Tällöin komponenttien toimintaa ei varsinaisesti laajenneta, vaan niitä säädetään erilaisilla konfiguraatioilla sovelluskehyksen asettamissa rajoissa. Esimerkiksi sovelluskehys voi olla toteutettu *tehdas*-suunnittelumallia [15] käyttäen, jolloin sovelluksen instantioimien olioiden tyyppi ja toiminnallisuus voidaan määritellä esimerkiksi konfiguraatiotiedostossa.

Plugin-kehyksissä sovelluskehys tarjoaa määritellyn rajapinnan, jonka sovelluskohtaisten komponenttien on toteutettava. Esimerkiksi uuden komponentin on toteutettava tietyt luokan metodit, jotta se toimisi oikein kehyksen osana. Plugin-kehyksissä uudet komponentit voidaan rekisteröidä automaattisesti kehyksen suorituksen osaksi esimerkiksi lataamalla ne ennalta määritetystä hakemistosta sovelluksen käynnistyksen yhteydessä. Näin sovelluksen laajentaminen on suhteellisen helppoa, sillä erikoistamisyksiköt *pluginit* voidaan sijoittaa tiedostoina kyseiseen hakemistoon.

Usein sovelluskehys ei kuulu selkeästi vain yhteen mainituista kehystyypeistä vaan sen eri osat erikoistetaan eri tavoin. Erikoistamismekanismia valittaessa voidaan ottaa huomioon sovelluksen osan muuntelutarve. Esimerkiksi jos variaatiot ovat pieniä tai helposti ennakoitavia, voi koottavan kehyksen mekanismi olla tilanteeseen sopiva. Toisaalta hankalammin ennalta arvioitavan erikoistamisen tapauksessa voi olla toivotumpaa käyttää muunneltavan kehyksen tai plugin-kehyksen mekanismeja [17].



Kuva 6. *Tietovuoarkkitehtuurin perusrakenne a) liukuhihna-arkkitehtuurina ja b) haarautuvana tietovuona*

3.2 Tietovuoarkkitehtuuri

Tietovuoarkkitehtuuri on arkkitehtuurityyli, jota yleensä käytetään tietovirtaa käsittelevissä järjestelmissä, joissa datalle suoritetaan sarja itsenäisiä, toisistaan riippumattomia laskentaoperaatioita [6]. Arkkitehtuuri koostuu prosessointiyksiköistä ja väylistä. Yksiköiden tehtävänä on lukea syötedataansa ja prosessoida siitä oma tulostevirta. Väylien toteutus yhdistää prosessointiyksiköt toisiinsa ketjuksi, jossa prosessointiyksiköt lukevat niitä edeltävien prosessointiyksiköiden tuottamaa tietovirtaa. Tämä mahdollistaa monimutkaisten tiedonkäsittelyprosessien toteuttamisen yksittäisinä helposti ymmärrettävinä komponentteina. Prosessointiyksiköiden muodostama tietovirta saattaa myös haarautua (kuva 6, kohta b), mutta yleisin tietovuoarkkitehtuurin ilmentymä on *liukuhihna-arkkitehtuuri*, jossa tietovirta etenee suoraviivaisesti haarautumatta [17].

Jokainen prosessointiyksikkö on itsenäinen komponentti, joka ei ole riippuvainen muusta kuin syötevirrastaan. Näin tietovuon prosessointiyksiköillä ei ole esimerkiksi jaettua tilaa, eivätkä ne ole tiukasti sidottuja muihin prosessointiyksiköihin. Tämä itsenäisyys mahdollistaa tietovuon muuntelemisen suhteellisen vapaasti, sillä uusien prosessointiyksiköiden tarvitsee vain toteuttaa yhteiseksi määritelty syöte- ja tulosteformaatti. Tietovuoarkkitehtuuri toteuttaa näin luvussa 2 esitellyistä taktiikoista *muutosten lokalisoinnin*, *tulevaisuuden muutosten odottamisen*, *kommunikaatioreittien rajoittamisen* ja *tiedon piilottamisen*. Nämä taktiikat mahdollistavat prosessointiyksiköiden lisäämisen tietovuohon aiheuttamatta värevaikutuksia muualle sovellukseen tehden siten uusien laskentayksiköiden lisäämisen olemassaolevaan tietovuohon yksinkertaiseksi.

Tietovuoarkkitehtuurin toinen etu on sen luonnollinen rinnakkaistettavuus. Itsenäi-

set prosessointiyksiköt voidaan sijoittaa erillisiin rinnakkain ajettaviin säikeisiin tai prosesseihin. Tällöin väylien täytyy toteuttaa rinnakkaista laskentaa tukeva viestinvälitysmekanismi puskureilla eri tahtiin tietoa prosessoivien prosessointiyksiköiden toiminnan mahdollistamiseksi.

Tietovuoarkkitehtuurit eivät sovellu interaktiivisiin järjestelmiin, sillä tietovirran välivaiheet ovat harvoin loppukäyttäjälle kiinnostavia tai edes järkevästi esitettävissä. Toisaalta tietovuoarkkitehtuurilla voidaan toteuttaa interaktiivisen järjestelmän ei-interaktiivinen osa, esimerkiksi videon muokkaussovelluksen operaatiot, jotka suoritetaan lopulliselle videolle käyttäjän antamien parametrien perusteella. *Kairan* toimintaperiaate ei ole erityisen interaktiivinen, joten näiden tietovuoarkkitehtuurin heikkouksien ei katsottu haittaavan sovelluksen käyttöä.

Tietovuoarkkitehtuuri voi myös johtaa suorituskykyongelmiin prosessointiyksiköiden standardoidun tietoformaatin vuoksi. Riippuen tietoformaatista saattaa jokainen prosessointiyksikkö joutua ensin muuntamaan sille välitetyn syötteen ohjelmointikielen ymmärtämiksi tietorakenteiksi. Prosessoidut tietorakenteet joudutaan muuntamaan takaisin standardoituun tietoformaattiin [17].

4. ARKKITEHTUURIN TOTEUTUS

Kairan arkkitehtuurin tavoitteena on tarjota kehys tiedonlouhinta-algoritmien ajamiseen matrikkelikirjasarjoille. Tutkimusprojektissa tavoitteena on irrottaa dataa useammista kirjasarjoista tutkimustyötä varten, joten *Kairan* arkkitehtuuri toteutettiin sovelluskehyksenä. Tiedonlouhinta tekstistä toteutetaan louhintakomponenteilla, jotka sovelluskehys ajaa tietovuoarkkitehtuurille tyypillisenä liukuhihnana. Liukuhihnan suorituksen jälkeen louhintakomponenttien ulostulosta koostetaan rakenteinen JSON-dokumentti, joka sisältää kunkin kirjan henkilön matrikkelitekstistä irrotetun datan. Sovelluskehyksen tehtävänä on tarjota rajapinta louhintakomponenttien toteuttamiselle ja niiden ajamiselle useille toisistaan poikkeaville lähdeaineistoille. Louhintakomponentteja voidaan lisätä ja muokata vapaasti kirjasarjojen tarpeen mukaan muuttamatta varsinaisen sovelluskehyksen toimintaa.

Sovelluskehysten etuna pidetään tavallisesti korkeampaa tuottavuutta ja pienempää kehitysaikaa koodin uudelleenkäytön ansiosta [23], mikä puolti sovelluskehyksen käyttöä *Kairan* toteuttamiseen. Jokaista erillistä kirjasarjan tiedonlouhintatoteutusta voidaan ajatella tuoteperheen jäsenenä. Tuoteperheen jäsenillä on yhteinen perusarkkitehtuuri.

Kairan sovelluskehyksen arkkitehtuuri on kehitetty iteratiivisesti ensimmäiseen kirjasarjaan tarkoitetun prototyypin pohjalta. Kehystä on refaktoroitu sisältämään kaikkien kirjasarjojen yhteiset toiminnallisuus- ja laatuvaatimukset ja uusia ominaisuuksia on lisätty tarpeen vaatiessa. Kehityksessä on sovellettu Martin Fowlerin nimeämää *HarvestedFramework*-mallia, jossa alkuperäisen sovelluksen pohjalta kehitetään toinen sovellus, ja näille yhteinen toiminnallisuus eriytetään sovelluskehykseksi. [10]

Kairan tapauksessa sovelluksen perustoiminnallisuus kehitettiin ensin *Siirtokarjalaisten tie* -kirjasarjalle ja myöhemmin sitä sovellettiin muun muassa *Suomen pienviljelijät* -kirjasarjaan. Vaikka tutkimusprojektin kannalta ensisijaisin kirjasarja koskettaa siirtokarjalaisia, muiden kirjasarjojen mukaan ottaminen helpotti sovelluskehyksen tarvitsemien ominaisuuksien tunnistamista ja suunnittelua. Tutkimusprojektin myöhemmissä vaiheissa myös muiden kirjasarjojen dataa on tarkoitus hyödyntää tutkimuksessa.

Kehykseen on sisällytetty ensisijaisesti toiminnallisuus, jota vaaditaan tiedonlouhintaprosessin ajamiseen ja rakentamiseen. Varsinaiset tiedonlouhinta-algoritmit eivät

kuulu *Kairan* ydinarkkitehtuurin piiriin, sillä ne ovat toistaiseksi kirjasarjakohtaisia ja siten toteutettu sovelluskehyksen erikoistamisrajapinnan kautta erillisiksi louhintakomponenteiksi. Poikkeuksena tästä on pieni joukko louhintakomponentteja, jotka ovat sovellettavissa suoraan tai kevyellä erikoistamisella useisiin kirjasarjoihin. Näiden komponenttien voidaan kuitenkin katsoa muodostavan oman erillisen moduulinsa, eivätkä ne siten varsinaisesti kuulu itse sovelluskehyksen rakenteeseen.

Tässä luvussa kuvataan ensisijaisesti *Kairan* sovelluskehyksen toteutusta, sen kehitysprosessia ja suunnitteluvalintoja. Tiedonlouhinta-algoritmeja ei kuvata tässä työssä tarkemmin muutoin kuin esimerkkeinä arkkitehtuurin suunnittelupäätöksiin vaikuttaneista tekijöistä tai havainnollistamaan arkkitehtuurin soveltamista.

4.1 Sovelluskehyksen tärkeimmät laadulliset vaatimukset

Sovelluskehyksen arkkitehtuurissa tehtyjen suunnittelupäätösten ymmärtämiseksi on ensin lyhyesti käsiteltävä sovelluskehykselle asetetut tärkeimmät vaatimukset. Vaatimusten pohjalta voidaan tunnistaa kehykselle oleellisimmat taktiikat, jotka kehyksen tulisi toteuttaa.

Ensimmäinen ja todennäköisesti tärkein vaatimus kehykselle on sen helppo ja nopea laajennettavuus. Tämän vaatimuksen on toteuduttava kahdella tasolla: kirjasarjaa käsittelevään koodiin on oltava helposti lisättävissä uusia dataa louhivia komponentteja. Toisekseen uusia kirjasarjoja käsitteleviä liukuhihnarakenteita on pystyttävä määrittelemään joustavasti.

Laajennettavuuden toteutumiseksi sovelluskehykseen on sovellettava siten erityisesti *muutosten lokalisointia* ja *tulevaisuuden muutosten odottamista*. Lisäksi laajennettavuuden kannalta oleellista on minimoida värevaikutukset, joten arkkitehtuurissa on sovellettava vielä erityisesti *tiedon piilottamista* ja *kommunikaatioreittien rajoittamista* riippuvuuksien hallitsemiseksi.

Toinen oleellinen vaatimus on louhintakomponenttien uudelleenkäytettävyys. Kirjoitettuja komponentteja on pystyttävä käyttämään eri liukuhihnan osissa eri tavoin parametrisoituina tai jopa useissa eri kirjasarjoissa. Tämä uudelleenkäytettävyys asettaa komponentille vaatimuksia, joita voidaan käsitellä *moduulin yleistämis* taktiikoina.

Näiden vaatimusten lisäksi oleellinen vaatimus on yksittäisten komponenttien testattavuus ohjelmiston oikeellisen toiminnallisuuden varmistamiseksi. Vaatimusten ja niihin liittyvien taktiikoiden toteutus on kuvattu tarkemmin sovelluskehyksen toiminnan ja rakenteen käsittelyn yhteydessä.

4.2 Sovelluskehyksen rakenne

Kairan sovelluskehyksen muunneltavuuden perusta rakentuu tietovuoarkkitehtuurille, jossa suurin osa muunneltavuudesta toteutetaan liukuhihnan louhintakomponenttien avulla. Louhintakomponentit ovat sovellusyksiköitä, jotka lukevat niille välitetyn tekstimateriaalin ja irrottavat (*"extract"*) siitä jonkin halutun datan kappaleen. Näin kuhunkin tekstistä irrotettavaan dataan liittyvä logiikka on eristetty omaan ohjelmistokomponenttiinsa lokalisoiden näin mahdollliset muutokset logiikkaan sovelluksen tulevaisuudessa.

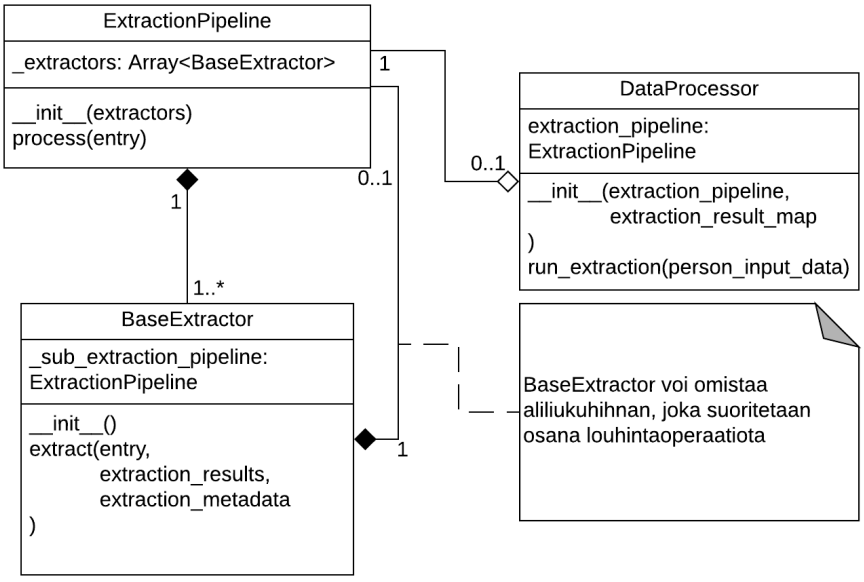
Esimerkki louhintakomponentista on komponentti, joka etsii tekstistä henkilön syntymäpäivän ja tallentaa sen erillisenä tietona tulosjoukkoonsa. Kukin louhintakomponentti vastaanottaa edellisten komponenttien tuottaman dataobjektin ja lisää siihen omat tuloksensa erillisen avaimen alle. Näin louhintakomponentit suorituksen edetessä tuottavat lopulta objektin, jonka sisältämä data voidaan tallentaa JSON-dokumenttiin myöhempää jatkokäsittelyä varten.

Sovelluksen toimintaa erikoistetaan toteuttamalla kullekin kirjasarjalle soveltuvia louhintakomponentteja, jotka toteuttavat liukuhihnan vaatiman rajapinnan. Ajon aikana sovelluskehys rakentaa konfiguraatiotiedoston perusteella liukuhihnan louhintakomponenteista ja ajaa sen jälkeen lähdemateriaalin jokaisen henkilön tiedot liukuhihnan prosessoitavaksi. Louhintakomponenttien ajamisessa käytetään sovelluskehyksille tyypillistä *Hollywood periaatetta*, jossa sovelluskehys huolehtii yksittäisten louhintakomponenttien kutsumisesta.

4.2.1 Liukuhihna ja louhintakomponentit

Kuvassa 7 on yksinkertaistettu UML-kaavio liukuhihnan toiminnalle tärkeimmistä komponenteista. Louhintakomponentit erikoistetaan periyttämällä ne liukuhihnan rajapinnan vaatimukset täyttävästä *BaseExtractor*-luokasta. *BaseExtractorin* tärkeimmät metodit ovat sen rakentaja ja *extract*-metodi. Suorituksen aikana sovelluskehys kutsuu *extract*-metodia välittäen louhintakomponenttiin tekstidatan (*entry* parametri), aiempien louhintakomponenttien tulosobjektin (*extraction_results*) ja datan irrotukseen liittyvää metadataa.

Louhintakomponentit ovat mahdollisia alustusparametreja lukuunottamatta tilattomia elinkaarensa aikana. Sovelluksen käynnistyessä liukuhihna ja sen komponentit instantioidaan ja samoja instansseja käytetään kaikkien datajoukon henkilöiden prosessointiin. Näin vältytään turhalta louhintakomponenttien uudelleen luomiselta ja tuhoamiselta. Ajon aikana liukuhihnan tai sen osien tila ei muutu, ainoastaan sen käsittelemän datan tulosobjektin sisältö muuttuu kun siihen lisätään uutta dataa. Vaikka tilattomuus hieman monimutkaistaa komponenttien rajapintaa vaatien

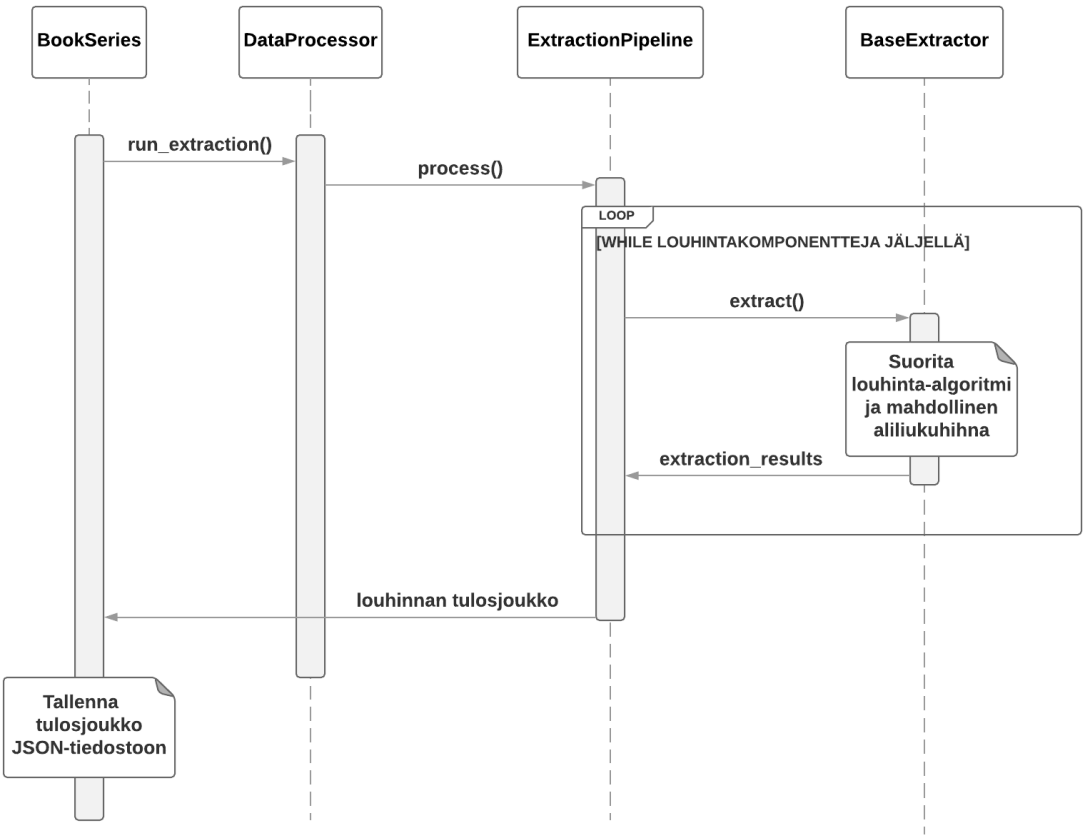


Kuva 7. *Tietovuon koostavat luokat.*

kaiken tarvittavan tiedon välittämisen *extract*-kutsun yhteydessä, todettiin sen helpottavan sovelluksen ylläpitoa ja testausta komponentin sisäisen tilan pysyessä koko suorituksen ajan samana. Tällöin komponentin tuloste on riippuvainen vain komponentin syötteestä ja mahdollisista alustusparametreista helpottaen esimerkiksi yksikkötestien kirjoittamista.

Liukuhihna luodaan instantioimalla luokka *ExtractionPipeline*, jolle välitetään tiedonlouhintakomponentit rakentajassa. Liukuhihna koostuu yhdestä tai useammasta louhintakomponentista. Liukuhihna ajaa sille annetun datan kutsuttaessa *process*-metodia ja palauttaa objektin, joka sisältää liukuhihnan komponenttien lähdetekstistä louhiman datan. Luokan tehtävänä on ensisijaisesti huolehtia louhintakomponenttien ketjuttamisesta ja mahdollistaa useiden louhintakomponenttien ajaminen kerralla. Luokka on ajon aikana tilaton alustusparametrejaan lukuunottamatta, joiden pohjalta liukuhihnan rakenne määritellään. Luokan *process*-metodissa kutsutaan louhintakomponenttien *BaseExtractorin* rajapinnan *extract*-metodia välittäen sille tekstin sekä liukuhihnan aikaisempien komponenttien tuottaman tulosjoukon ja metadatan.

Kirjasarjakohtaisen liukuhihnan suoritusta on havainnollistettu sekvenssikaaviossa kuvassa 8. *DataProcessor* huolehtii datajoukon syöttämisestä kirjasarjatasolla pääliukuhihnalle. Kun sovellukseen on ladattu kirjasarjan datan sisältävä XML, *DataProcessor* syöttää datajoukon kunkin henkilön tekstikappaleen yksitellen kirjasarjan liukuhihnalle. Liukuhihnan tuottama data tallennetaan listaan, joka tallennetaan JSON-dokumentiksi, kun koko datajoukko on käsitelty. Sekvenssikaaviossa näkyvä *BookSeries*-luokka tarjoaa kirjasarjakohtaiset alustusoperaatiot ja toimii korkeim-



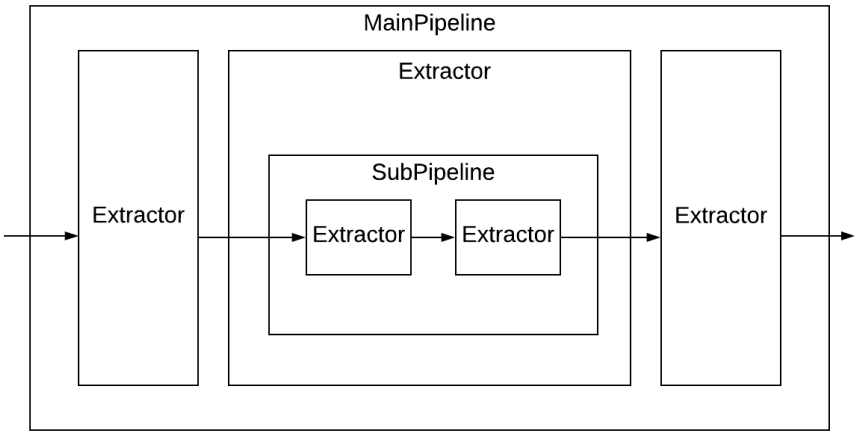
***Kuva 8**. Louhintaprosessi koostuu kirjasarjalle rakennetusta liukuhihnasta, jonka louhintakomponentit suoritetaan sarjassa.*

man tason suorittavana komponenttina. Palaamme tarkemmin *BookSeries*-luokan toimintaan aliluvussa 4.4.2.

DataProcessor suunniteltiin tulevaisuuden mahdolliseksi variaatiopisteeksi datan syöttämiselle järjestelmään. Tätä mahdollisuutta hyödynnettiin, kun järjestelmään lisättiin tuki luonnollisen kielen lauserakenteita kuvaavalle CoNNL-U formaatille. Kun syötetiedostona toimivaan XML-tiedostoon lisättiin raakatekstin lisäksi CoNNL-U formaatti, kasvoi datatiedostojen koko merkittävästi. Tämän myötä *Kairan* datatiedoston luku muutettiin kerralla muistiin luettavasta prosessista syötetiedostoa virtana käsitteleväksi prosessiksi. Tämä muutos oli tehtävissä varsin pienillä muutoksilla kehykseen, sillä datan syöttäminen liukuhihnalle oli toteutettu yhteen paikkaan ja siten muutokset syötedatan formaattiin abstrahoitu liukuhihnakomponenteilta.

4.2.2 Aliliukuhihnat

Usein tekstissä on datakokonaisuuksia, joiden poimiminen tekstistä vaatii monimutkaista käsittelyä, josta osa on toteutettavissa aiemmin toteutetuilla komponenteilla, kun taas osa vaatii uusia louhintakomponentteja. Esimerkki tällaisesta kokonaisuu-



Kuva 9. *Tietovuon liukuhihnan ja irrottajakomponenttien kokonaisuus. Louhintakomponentilla voi olla aliliukuhihna.*

desta kirjasarjoissa on kirjan henkilöiden puolisoiden tiedot. Yleensä puolisoille on tekstissä lueteltu ainakin heidän nimensä, entinen sukunimensä ja syntymätiedot. Osaan näistä tiedoista, esimerkiksi syntymäajan poimintaan tekstistä voidaan käyttää samaa logiikkaa eri alustusparametreilla kuin tekstin ensisijaisen henkilön datan poimintaan.

Kuvassa 9 on esitetty sovelluksen liukuhihnan rakenne, jossa toisella louhintakomponentilla on oma aliliukuhihnansa. Kuvan *SubPipeline* on tavallinen aiemmin esitelty *ExtractionPipeline*-luokka. Liukuhihnan ajamisesta eli *process()*-kutsusta vastaa kuitenkin louhintakomponentti, jonka osana aliliukuhihna on.

Käytännössä aliliukuhihnat ovat vain sovelluskehittäjän avuksi suunniteltu rakenne ja louhintaprosessin voi edelleen ajatella suoritettavan yksitasoisena liukuhihnana, jossa louhintaoperaatioita suoritetaan sarjassa. Rakenne kuitenkin lisää joustavuutta koodin uudelleenkäyttöön. On esimerkiksi mahdollista käsitellä aliliukuhihnalle syötettävää tekstiä ennen sen välittämistä aliliukuhihnan käsiteltäväksi ilman, että nämä tekstin muutokset välittyisivät pidemmälle liukuhihnalla. Aliliukuhihnalle voidaan syöttää vain tarvittava osa henkilökuvauksen tekstistä. Esimerkiksi teksti, joka sisältää vain puolisoa koskevat tiedot helpottaen oikean datan löytämistä tekstistä. Tällöin sekä ensisijaisen henkilön että puolison datan irrottamiseen voidaan käyttää samaa toteutusta ilman, että louhintakomponentissa itsessään tarvitaan logiikkaa erottelemaan löydetyn datan kohdehenkilöä. Yksinkertaisena esimerkkinä syntymäpäivän etsivä komponentti voi tulkita ensimmäisen löydetyn päivämäärän kohdehenkilönsä syntymäpäiväksi, sillä aliliukuhihnalle välitetty tekstidata sisältää vain puolison syntymäpäivän. Kun aliliukuhihna on suoritettu, palauttaa sen isäntäkomponentti aliliukuhihnan tulokset pääliukuhihnalle, joka jatkaa suoritusta jälleen koko tekstimateriaalilla.

Toinen ohjelmoijan kannalta aikaa säästävä aliliukuhihnojen ominaisuus on suurempien uudelleenkäytettävien louhintakomponenttien rakentaminen kompositiolla yksinkertaisemmista komponenteista. On esimerkiksi mahdollista määrittää komponentti, jolla on aina tietynlainen aliliukuhihna, jonka toteutusta ei tarvitse erikseen konfiguroida.

4.3 Liukuhihnan konfigurointi

Kirjasarjojen liukuhihnojen operaatiot määritellään YAML-merkintäkielellä kirjasarjakohtaisesti. YAML valittiin kieleksi konfiguraatioiden määrittelyyn sen helpon luettavuuden vuoksi. Lisäksi YAML tukee viitteiden määrittelyä dokumentin elementtien välille sekä kommenttien lisäämistä konfiguraation joukkoon. Nämä ominaisuudet tekivät siitä esimerkiksi JSONia soveltuvamman formaatin sovelluksen konfigurointiin.

Jokaisen kirjasarjan hakemistoon määritellään *pipeline_ config.yaml*-tiedosto, joka sisältää koko kirjasarjan liukuhihnan kuvauksen. Listauksessa 2 on ote *Siirtokarjalaisten tie* -kirjasarjalle määritellystä liukuhihnasta. Suunnitteluperiaatteena on ollut erottaa tiedonlouhinnan toteutus tiedonlouhintaoperaation kuvauksesta. Toisin sanoen louhintalogiikan toteutus on toteutettu Python-koodilla, kun taas näiden louhintakomponenttien määritteleminen isommaksi ajettavaksi kokonaisuudeksi on määritelty YAML-merkintäkielellä. Tämä todettiin käytännön kannalta toimivaksi ratkaisuksi, sillä YAML tarjosi yhtenäisen standardin tavan määritellä sovelluksen rakenne eri kirjasarjoille.

Liukuhihnan rakenteen muutokset rajoittuvat yleensä konfiguraatiotiedostoon, mikä ehkäisee muutostarpeita muualla sovelluksessa. Standardi tapa rakentaa liukuhihna myös rajoittaa mahdollisuuksia määrittää liukuhihnan rakenne "väärin". Toisin sanoen se estää mielivaltaisten *kommunikaatioväylien* määrittämisen liukuhihnan osien välille pakottaen siten järjestelmään yhdenmukaisen tiedonvälitysmallin. Kommunikaation rajoittaminen on taktiikka, jolla pyritään ehkäisemään värevaikutusten syntymistä sovellusta muutettaessa jälkeenpäin.

```
1 pipeline:
2   - !Extractor &PrimaryPersonName {
3       module: "name_extractor",
4       class_name: "NameExtractor",
5       options: {
6         output_path: "primaryPerson"
7       }
8     }
9
```

```
10   - !Extractor {
11       module: "common.extractors.bool_extractor",
12       class_name: "BoolExtractor",
13       options: {
14         patterns: {
15           animalHusbandry: 'karjanhoitoa?\b|karjatalous\b',
16           dairyFarm: 'lypsy-|lypsy\b|lypsykarja(?!sta)',
17           maanhankintalaki: '(?:maanhankinta){s<=1,i<=1}',
18           coldFarm: 'kylmät'
19         }
20       }
21
22   - !Extractor {
23       module: "migration_route_extractors",
24       class_name: "MigrationRouteExtractor",
25       options: {
26         output_path: "primaryPerson"
27       },
28       pipeline: [
29         !Extractor {
30             module: "migration_route_extractors",
31             class_name: "KarelianLocationsExtractor"
32         },
33         !Extractor {
34             module: "migration_route_extractors",
35             class_name: "FinnishLocationsExtractor"
36         }
37       ]
38     }
```

Listaus 2. *Lyhennetty näyte kirjasarjan liukuhihnan konfiguraatiotiedostosta.*

Konfiguraatiotiedosto alkaa ylimmän tason liukuhihnan määrittelyllä avainsanalla *pipeline*, joka on lista louhintakomponentteja, jotka määritellään YAML-objektilla *!Extractor*. Käynnistyksen yhteydessä *Kaira* etsii kirjasarjakohtaiset määrittelytiedostot ja jäsentää konfiguraatiotiedostot *PyYAML*-kirjaston [22] avulla. Jäsennyksen yhteydessä sovellus instantioi *!Extractor*-määritykset Python-olioiksi käyttäen konfiguraatiossa tarjottuja luokka- ja moduulinimiä. Luoduista olioista rakennetaan valmis ajettavissa oleva liukuhihnarakenne, jonka läpi tekstidata voidaan ajaa.

Louhintakomponenttien konfigurointi

Sovelluksen rakentaessa liukuhihnan YAML-konfiguraation listauksesta voidaan kullekin komponentille välittää siinä määriteltyjä lisäasetuksia. Tuetut ja vaaditut lisäasetukset vaihtelevat louhintakomponentista riippuen. Esimerkkilistauksessa 2 listattu *BoolExtractor* on yleiskäyttöinen louhintakomponentti, joka etsii säännöllis-

ten lausekkeiden perusteella sanoja tekstistä tallentaen tulokseksi totuusarvon. Tälle louhintakomponentille on siten määriteltävä toivotut säännölliset lausekkeet sekä avaimet, joiden alle lipun arvo tallennetaan. Näin uusien totuusarvojen lisääminen ei vaadi muutoksia varsinaiseen ohjelmakoodiin, sillä erikoistaminen voidaan suorittaa parametrisoinnilla konfiguraatiosta käsin koottavien kehysten tapaan. Tämä mahdollistaa samojen louhintakomponenttien käyttämisen myös useissa osissa liukuhihnaa tai eri kirjasarjoissa. Esimerkin *BoolExtractor*-komponenttia voitaisiin käyttää useissa paikoissa liukuhihnaa irrottamaan dataa eri aihepiireihin ryhmiteltyinä.

Liukuhihnan konfiguraatiossa voidaan myös kuvata kunkin louhintakomponentin mahdollinen aliliukuhihna avainsanalla *pipeline*. Tämä louhintakomponentin attribuutti on samankaltainen lista louhintakomponentteja kuin konfiguraatiotiedoston juuritason objekti. Esimerkkilistauksen *MigrationRouteExtractor*-komponentti koostuu aliliukuhihnalle määritellyistä komponenteista *KarelianLocationsExtractor* ja *FinnishLocationsExtractor*. Aliliukuhihnan komponentteja voidaan parametrisoida ja erikoistaa samaan tapaan kuin päätason komponentteja ja niillä voi vuorostaan olla omia aliliukuhihnojaan ja komponenttejaan. Tämä rekursiivinen rakenne mahdollistaa suhteellisen joustavan ja hienosyisen louhintaoperaatioiden määrittelemisen ja komponenttien uudelleenkäytön ilman uuden ohjelmakoodin kehittämistä.

4.4 Louhintakomponenttien väliset riippuvuudet

Suurin osa louhintakomponenteista on itsenäisiä yksiköitä, jotka tarvitsevat toimiakseen vain raakatekstin, josta irrottaa dataa mahdollisten alustusparametrien määrittelemällä tavalla. Ajoittain louhintaprosessissa kuitenkin tarvitaan päättelyä, johon on hyödynnettävä aiempien louhijoiden irrottamaa dataa. Otetaan esimerkiksi listauksen 3 kuvitteellinen tekstikappale, jossa ensisijaisena henkilönä on mies, ja puolisona hänen vaimonsa.

```
1   SIMAKUUTIO, AHTO maanviljelijä, synt. 10. 3. -92 Hiitolassa Puol.
       Essi o.s. Esimerkki, synt. 10. 3. 1900 Hiitolassa.
2   Isäntä haavoittui sodassa jonka aikana hän oli ylikersantti. Rouva
       oli sodan aikana mukana lotta-toiminnassa.
```

***Listaus 3**. Tekstikappale jossa ensijainen henkilö on mies ja puoliso on nainen.*

Haluamme irrottaa kappaleesta tiedot koskien Ahton sotilasarvoa ja tiedon rouvan osallistumisesta lotta-toimintaan. Oletetaan että käytössämme on louhintakomponentit, jotka tunnistavat tekstistä sanat "ylikersantti"ja "lotta-toiminta", joko perusmuodossa tai taivutettuina. Ongelmaksi muodostuu päätellä, kenelle nämä datan kappaleet kuuluvat, eli kuinka asettaa löytynyt ylikersantin arvo Ahton dataobjektin

```
1   {
2     "primaryPerson": {
3       "name": {
4         "firstNames": "AHTO",
5         "surname": "SIMAKUUTIO",
6         "gender": "Male"
7       },
8     "spouse": {
9       "name": {
10        "firstNames": "ESSI",
11        "surname": "SIMAKUUTIO",
12        "gender": "Female"
13      }
14  }
```

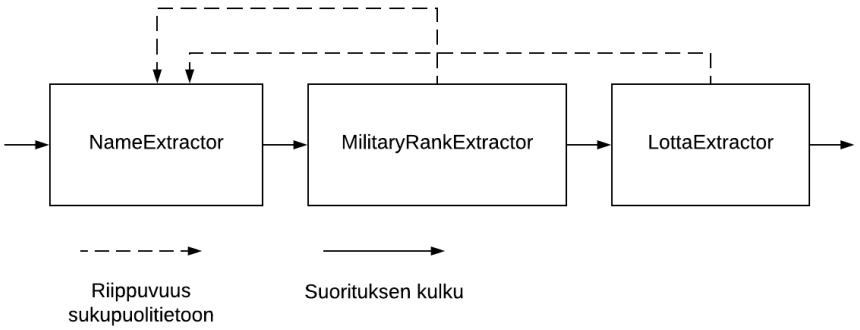
Listaus 4. *Tulosobjekti jossa tiedot ensisijaisen henkilön ja tämän puolison nimistä ja sukupuolesta.*

alle ja lotta-merkintä rouvan dataobjektin alle. Historiasta tiedämme, että tekstin sotilasarvo viitannee hyvin varmasti mieheen ja lotta-toiminta puolestaan naiseen, joten voimme käyttää tietoa henkilöiden sukupuolesta liittämään löydetyn datan oikealle henkilölle.

Siirtokarjalaisten tie -kirjasarjassa kuitenkin ensisijainen henkilö ei aina ole mies ja puoliso nainen, sillä tekstikappaleissa ensisijaisen henkilön aseman on saanut Karjalasta evakkoon lähtenyt henkilö. Kirjasarjan liukuhihnalla on kuitenkin louhintakomponentti, joka päättelee henkilön etunimistä tämän todennäköisen sukupuolen. Komponentti vertaa henkilön nimeä listoihin miesten ja naisten etunimistä merkiten siten ensisijaisen henkilön ja tämän puolison sukupuolet. Listauksessa 4 on esitetty liukuhihnan aiempien komponenttien luoma dataobjekti, jossa ensisijaisen henkilön ja tämän puolison dataobjekteihin on tallennettu heidän nimensä ja sukupuolensa.

Sotilasarvon ja lotta-toiminnan tekstistä irrottavat komponentit voisivat siis käyttää aiempien louhintakomponenttien tuottamaa tulosjoukkoa asettamaan tuloksensa oikean henkilön alle yksinkertaisella päättelyllä: Jos tekstistä löytyi sotilasarvo, katso kumpi dataobjektin henkilöistä on mies *gender*-attribuutin perusteella ja oleta sotilasarvon kuuluvan tälle henkilölle.

Nyt sotilasarvokomponentti on riippuvuussuhteessa (kuva 10) aiemmin ajettuun sukupuolen päättelevään louhintakomponenttiin. Toimiakseen se tarvitsee tämän komponentin tulosjoukosta tiedon ensisijaisen henkilön tai tämän puolison sukupuolesta, jotta se kykenisi päättelemään mihin tulosjoukossa sijoittaa louhimansa datan.



Kuva 10. *Louhintakomponentit voivat olla riippuvaisia aiemmin suoritettujen komponenttien tulosjoukoista. Kuvassa jälkimmäiset komponentit ovat riippuvaisia ensimmäisen tuottamasta sukupuolidatasta.*

4.4.1 Riippuvuuden välittäminen komponentille

Esimerkkimme sotilasarvokomponentti tarvitsee siis logiikkaansa dataa aiemmin tuotetusta louhinnan tulosjoukosta. Naiivi tapa välittää tämä data sotilasarvokomponentille olisi kirjoittaa sotilasarvokomponenttiin listauksen 5 mukainen toteutus, jolla tulosjoukko objektista *extraction_results* haetaan ensisijaisen henkilön sukupuoli.

```
1 pp_gender = extraction_results['primaryPerson']['name']['gender']
```

Listaus 5. *Joustamaton tapa hakea riippuvuuden data aiemmasta tulosjoukosta*

Tämä toteutus on kuitenkin useammalla tavalla ongelmallinen. Kovakoodattu polku hajoaa herkästi, mikäli mikä tahansa osa tulosjoukon rakenteesta muuttuu. Toiseksi toteutus esittelee kovakoodatun riippuvuussuhteen suoraan sotilasarvokomponentin toteutuksessa. Kaikki sotilasarvokomponentin instanssit yrittäisivät hakea sukupuolta samasta tulosjoukon osasta, mikä ei ole suotavaa komponentin uudelleenkäytettävyyden kannalta. On helposti kuviteltavissa tilanne, jossa esimerkiksi eri kirjasarjassa sukupuoli löytyy hieman erilaisesta paikasta tulosjoukko-objektia tai haluaisimme liukuhihnan toisessa sotilasarvokomponentin instanssissa hakea esimerkiksi puolison sukupuolen ensisijaisen henkilön sukupuolen sijaan.

Tarkemmin tarkasteltuna voimme todeta listauksen 5 rikkovan *muutosten lokalisointi* -taktiikkaa sekä lisäävän potentiaalia värevaikutuksille. Muutokset muissa komponenteissa helposti väreilevät tähän komponenttiin, sillä komponentti muodostaa riippuvuussuhteen epämuodollisesti toiseen komponenttiin. Tämä rikkoo *kommunikaatioreittien rajaus* -taktiikkaa. Näiden taktiikoiden toteuttamiseksi tarvitaan siten suunnitteluratkaisuja, jotka katkaisevat suoran riippuvuusketjun syntymisen muihin komponentteihin ja näin ehkäisevät värevaikutusten syntymistä myöhempien muutosten myötä.

```
1  pipeline:
2    - !Extractor &PrimaryPersonName {
3        module: "name_extractor",
4        class_name: "NameExtractor",
5        options: {
6          output_path: "primaryPerson"
7        }
8      }
9
10   - !Extractor {
11       module: "military_rank",
12       class_name: "MilitaryRankExtractor",
13       depends_on: [ *PrimaryPersonName ]
14   }
```

***Listaus 6**. Riippuvuussuhteen määrittely YAML-konfiguraatiotiedostossa.*

Louhijakomponentin riippuvuussuhteet on siis määriteltävä komponentin varsinaisen toteutuksen ulkopuolella ja ajonaikana välitettävä sille käyttöä varten. Tätä tekniikkaa kutsutaan tavallisesti riippuvuusinjektioksi *(dependency injection)*. Riippuvuusinjektiossa objekti, joka tarvitsee palvelua ei itse etsi tai rakenna tarvitsemaansa palvelua, vaan kyseinen objekti tai palvelu välitetään sille ulkopuolelta. Tekniikan tarkoituksena on ulkoistaa riippuvuudet komponentista ja siten vähentää komponentin tiukkaa riippuvuutta muihin ohjelmistokomponentteihin [11]. *Kairan* liukuhihnan tapauksessa komponentin toteutusta ei haluta tiukasti sitoa tulosjoukon tiettyyn data-alkioon, sillä samaa louhintakomponenttia voidaan haluta käyttää uudelleen eri data-alkioiden kanssa liukuhihnan muissa osissa.

Riippuvuussuhteet määritellään liukuhihnan konfiguraatiotiedostossa. Listauksessa 6 on esitetty konfiguraatio, jossa toinen louhintakomponentti edustaa esimerkkimme sotilasarvon etsivää louhijaa ja ensimmäinen henkilön nimen ja siitä sukupuolen päättelevää louhintakomponenttia. YAML mahdollistaa viitteiden määrittämisen dokumentissa olevien elementtien välille. Rivillä 2 ensimmäiselle komponentille määritellään alias *PrimaryPersonName* käyttäen &-notaatiota. Rivillä 13 sotilasarvolouhija listaa tarvitsemansa riippuvuudet ja viittaa määriteltyyn aliakseen *-notaatiolla.

Konfiguraatiossa julistettua riippuvuussuhdetta käytetään liukuhihnan rakentamisen aikana välittämään sotilasarvokomponentille viite aiempaan komponenttiin, jonka tuloksesta se on riippuvainen. Tämän viitteen avulla sotilasarvokomponentti voi louhintaprosessin aikana hakea tarvitsemansa datan tulosjoukosta.

Konfiguraatiotiedoston lisäksi ohjelmoijan on julistettava sotilaskomponentin toteutuksessa sen vaatimat riippuvuudet ja nimetä ne. Tämä tapahtuu yleensä louhintakomponentin rakentajassa listauksen 7 esittämällä tavalla.

```
1   def __init__():
2     super(MilitaryRankExtractor, self).__init__()
3     self._declare_expected_dependency_names(['name_data'])
4
5   def _extract():
6     gender = self._deps['name_data']['gender']
7     print('Gender␣of␣the␣person␣is', gender)
```

Listaus 7. *Riippuvuuksien julistaminen ja nimeäminen louhintakomponentin rakentajassa ja riippuvuuden sisältämän datan käyttäminen.*

Kutsumalla louhintakomponentin *_declare_expected_dependency_names* metodia ohjelmoija kertoo tämän louhintakomponentin odottavan yhtä ulkoista riippuvuutta määriteltäväksi konfiguraatiotiedostossa ja nimeävänsä kyseisen riippuvuuden "name_data":ksi. Riippuvuuksien nimet julistetaan samassa järjestyksessä, kuin ne on listattu konfiguraatiotiedoston *depends_on* listauksessa.

Louhintaprosessin suorituksen saapuessa *MilitaryRankExtractor*iin, se hakee taustalla tarvitsemansa riippuvuudet aiempien louhintakomponenttien tuloksista käyttäen tähän liukuhihnan alustusvaiheessa määriteltyä viitettä. Riippuvuus on objekti, joka sisältää aiemmin liukuhihnalla ajetun louhintakomponentin tulosjoukon. Haettu data asetetaan louhintakomponentin jäsenmuuttujaan nimeltä *_deps*. Tämä muuttuja on objekti, johon määritellyt riippuvuudet asetetaan niille *_declare_expected_dependency_names*-metodilla määriteltyjen avainten alle. Listauksessa 7 voidaan nähdä, kuinka riippuvuuden sisältämä data otetaan käyttöön *_extract* metodissa hyödyntäen rakentajassa määriteltyä riippuvuuden nimeä.

Edellä kuvattu malli riippuvuuksien julistamiseksi ja käyttämiseksi ratkaisee aiemmin kuvatut ongelmat liittyen komponentin uudelleenkäyttöön ja värevaikutuksiin. Sovelluskehys toimii välikappaleena eri louhintakomponenttien välillä injektoiden komponentin tarvitsemat riippuvuudet ajon aikana komponentin käyttöön. Menetelmä siirtää riippuvuussuhteiden määrittämisen sovelluskomponentista konfiguraatiotiedostoon lisäten komponentin uudelleenkäytettävyyttä. Ratkaisu myös toteuttaa *kommunikaatioreittien rajoitus* -taktiikan, sillä riippuvuuksien määrittely ja datan välitys komponentilta toiselle tapahtuu vain edellä kuvatulla tavalla. Menetelmä ei kuitenkaan ratkaise kaikkia mahdollisia riippuvuuden muotoja. Komponentti esimerkiksi odottaa edelleen riippuvuudeltaan tietyn formaatin dataobjektia. Nämä ongelmat ovat kuitenkin hyväksyttäviä rajoitteita sovelluksen tarpeet huomioiden.

4.4.2 Kirjasarjojen lisääminen plugineina

Kairan arkkitehtuurin yksi isoista suunnitteluperiaatteista on ollut suunnitella se helposti sovellettaviksi useisiin kirjasarjoihin. Lopultakin ohjelmistokehyksen tehtä-

vä on tarjota yhteinen kehysarkkitehtuuri useille samankaltaisille sovelluksille. Niinpä *Kairan* olisi kehyksenä tarjottava korkean tason modulaarinen rajapinta uusien kirjasarjojen tuen lisäämiselle ohjelmaan. Uusien kirjasarjojen lisäämisen sovellukseen ei tulisi vaikuttaa muiden kirjasarjojen toteutukseen. Vastaavasti yhden kirjasarjan toteutuksen muokkaamisen ei tulisi väreillä muiden kirjasarjojen toteutukseen. Nämä vaatimukset on täytetty kehittämällä *Kairaan* plugin-järjestelmä, jolla uusia kirjasarjoja voidaan lisätä sovelluskehyksellä ajettavaksi.

Pluginit voidaan määritellä tietyllä tavalla rakenteistetuiksi koodi- ja datakokoelmiksi, jotka vaikuttavat järjestelmän toiminnallisuuteen. Pluginit tarjoavat sovelluksen arkkitehtuuriin ennalta määritellyn laajennospisteen, jonka kautta sovelluksen toimintaa voidaan laajentaa [7].

Uuden kirjasarjan plugin koostuu pohjimmiltaan sille suunnitellusta liukuhihnamäärittelystä sekä useasta kirjasarjan tarvitsemasta louhintakomponenttien toteutuksesta. Uuden kirjasarjan toteutus ja konfiguraatiotiedostot sijoitetaan uuteen hakemistoon plugineille suunnitellun hakemiston alle ja yhdessä ne muodostavat kirjasarjan plugintoteutuksen ja rajapinnan. Sovelluksen käynnistyessä sovelluskehys tutkii plugin-hakemiston alihakemistoja etsien sieltä kirjasarjan määrittäviä *manifestitiedostoja*.

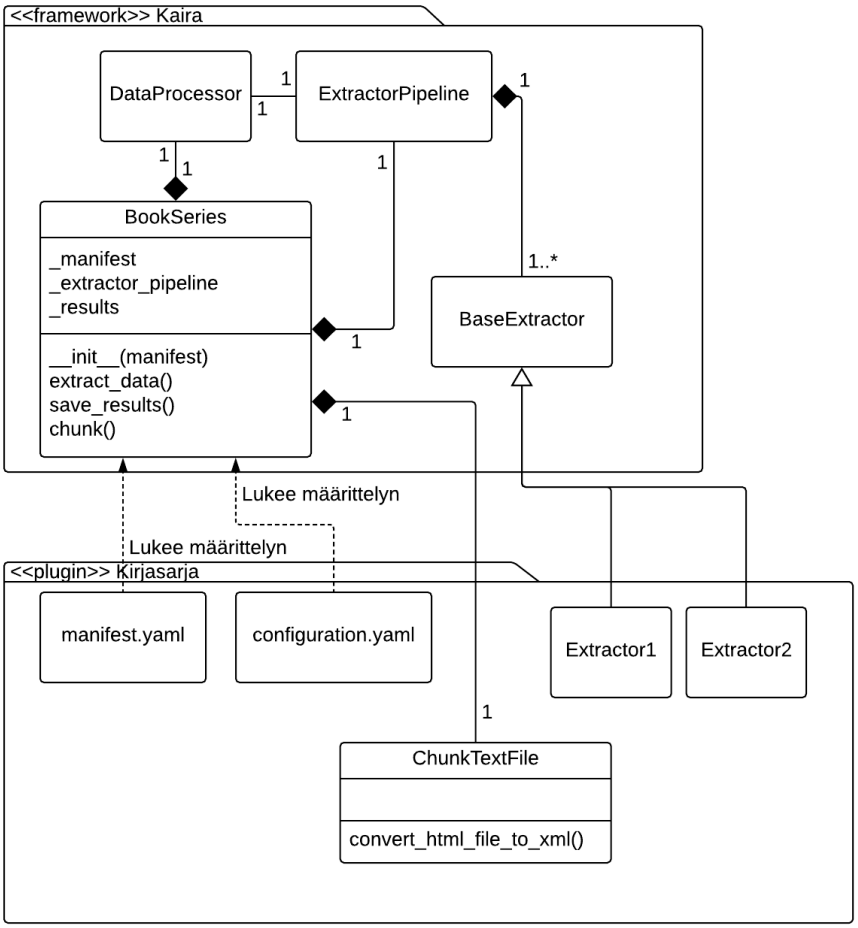
```
1 # Kirjasarjan tunniste, joka löytyy XML-tiedostojen <DATA> tagista
2 book_series_id: siirtokarjalaiset
3 # Polku moduuliin plugin-hakemistossa, joka sisältää HTML -> XML
     muunnos logiikan
4 chunker: chunktextfile.py
```

***Listaus 8**. Siirtokarjalaisten tie -kirjasarjan manifestitiedosto.*

Manifestitiedostot ovat yksinkertaisia YAML-tiedostoja, jotka sisältävät muun muassa kirjasarjan tunnisteen sekä mahdollisesti tietoa kirjasarjalle erityisten ohjelmistomoduulien poluista niiden instantioimista varten. Listauksessa 8 on kuvattu *Siirtokarjalaisten tie* -kirjasarjan manifestitiedosto.

Kirjasarjan tunniste on sama merkkijono, joka on löydettävissä XML-datatiedostojen juurisolmusta. *Kairan* käynnistyessä se etsii kaikki manifestitiedostot hakemiston *extractors/bookseries/* alihakemistoista ja päättelee niissä listatuista tunnisteista, mitkä sille syötettävistä XML-datatiedostoista ovat tuettuja. Mikäli syötetyn XML-tiedoston tunniste vastaa jotain manifestitiedostojen tunnistetta, sovelluskehys lataa kyseisen kirjasarjan liukuhihnakonfiguraation ja rakentaa kirjasarjalle ominaisen liukuhihnan tekstin lukemista varten.

Kuvassa 11 on havainnollistettu *Kairan* plugin-järjestelmän rakennetta. *BookSeries*-luokka huolehtii kirjasarjapluginin lataamisesta ja sen käyttämisestä. Luokan



***Kuva 11**. Plugin-järjestelmän rakenne*

rakentajassa sille välitetään aiemmin ladattu kirjasarjapluginin manifestitiedosto, jonka sisältämän datan pohjalta luokka on valmis suorittamaan kirjasarjapluginilla operaatiot *extract_ data, save_ results* ja *chunk*.

Kun käyttäjä valitsee komentoriviparametreilla jonkin näistä komennoista suoritettavaksi tietylle kirjasarjalle, *BookSeries*-luokan komentoa vastaavaa metodia kutsutaan. Esimerkiksi *extract_ data*-metodi lukee manifestitiedoston perusteella kirjasarjan liukuhihnan konfiguraatiotiedoston ja rakentaa sen pohjalta kirjasarjan liukuhihnan ja lopuksi ajaa tekstimateriaalin liukuhihnan läpi käyttäen *DataProcessor*-luokkaa.

Kuvasta 11 voidaan havaita, että kaikki pluginliukuhihnan louhintakomponentit on periytetty kehyksen *BaseExtractor*-luokasta. HTML-tiedostojen muuttamiseksi *Kairan* analysoitavissa olevaan XML-formaattiin kirjasarjapluginin on myös tarjottava moduuli, joka sisältää funktion *convert_ html_ file_ to_ xml*. Tämän funktion moduulin sijainti on kuvattu manifestitiedostossa attribuutilla *chunker*. *BookSeries*-luokka lataa moduulin dynaamisesti kun käyttäjä pyytää sitä suorittamaan *chunk*-

operaation HTML-tiedostolle.

Edellä mainittujen rajapintavaatimusten lisäksi sovelluskehyksellä ei ole kirjasarjapluginille muita vaatimuksia. Kirjasarjaplugini voi siis sisältää mitä tahansa tarvitsemiaan palveluita, apumoduuleita ja muuta toteutusta, joita sen louhintakomponentit voivat käyttää omassa toteutuksessaan. Tällöinkin kuitenkin suoritusta hallitsee aina sovelluskehys, joka huolehtii pluginin lataamisesta ja suorittamisesta rajapinnan kautta. Täten tämän plugin-rajapinnan kautta ei ole mahdollista muokata itse sovelluskehyksen toteutusta tai laajentaa sovelluskehyksen ydintä.

4.4.3 Useat rinnakkaiset tekstiformaatit syötedatana

Suuri osa *Kairan* tiedonlouhintalogiikasta tutkii raakatekstiä etsien dataa rakenteettomasta tekstimassasta. On kuitenkin olemassa edistyneitä tekstin analysointimenetelmiä, jotka hyödyntävät erityisiä tekstiformaatteja.

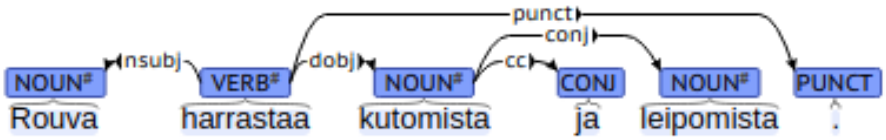
Yksi tällainen tekstiformaatti on *CoNLL-U*, joka kuvaa tekstissä esiintyvien lauseiden sanojen välisiä suhteita perustuen sanojen sanaluokkiin. Formaatti tuotetaan merkitsemällä tekstin sanoille niiden sanaluokat ohjelmallisesti, jonka jälkeen kielioppisääntöjen mukaisesti voidaan määritellä sanojen väliset suhteet lauseessa [5]. Esimerkki tällaisesta suhteesta on predikaatin, subjektin ja objektin suhde. Jos tekstistä esimerkiksi tunnistetaan verbi, voidaan *CoNLL-U*-formaatin avulla päätellä verbin tekijä ja tekemisen kohde.

Kairan käyttötapauksessa formaatti osoittaa hyödyllisyytensä esimerkiksi *Siirtokarjalaisten tie* -kirjasarjan tekstien loppuosien vapaamman muotoisen tekstin analysoinnissa. Useimmissa kirjasarjan teksteissä on pariskunnan molempien henkilöiden kuvauksia, esimerkiksi heidän harrastuksensa. Haasteena on kuitenkin tunnistaa tekstistä kenelle harrastus kuuluu esimerkiksi lauseesta:

Rouva harrastaa kutomista ja leipomista. Isäntä harrastaa hevosurheilua ja kalastusta.

Yksinkertainen harrasteisiin liittyvien sanojen etsiminen tekstistä ei kerro keneen nämä sanat tekstissä liittyvät, joten tällaisten suhteiden päättelyyn on hyödynnettävä lauserakenteen kuvaavaa *CoNLL-U* formaattia. Formaatilla voidaan tietyt harrastesanat lauseesta yhdistää joko sanaan rouva tai isäntä. Kuvassa 12 on havainnollistettu *CoNLL-U* formaatin kuvaamaa lauserakennetta, jossa ohjelmisto on tunnistanut sanaluokat ja päätellyt niistä näiden väliset suhteet.

Kaira tuottaa *CoNLL-U*-formaatin myöhempää käyttöä varten Turun Yliopiston kehittämällä *Finnish-dep-parser*-työkalulla [16], joka muuntaa raakatekstin *CoNLL-U* -formaattiin. Prosessi on hidas suurelle tekstimassalle, joten se ajetaan kirjasar-



Kuva 12. *Havainnekuva CoNLL-U-formaatin kuvaamasta lauserakenteesta. Kuva on tuotettu Finnish-dep-parserin tuottamasta CoNLL-U datasta.*

```
<DATA bookseries="siirtokarjalaiset" book_number="2">
  <PERSON name="TUOKKO, MATTI" approximated_page="507-509">
    <RAW>
      ...
    </RAW>
    <CONLLU>
      ...
    </CONLLU>
  </PERSON>
</DATA>
```

Listaus 9. *XML-dokumentin rakenne, jossa henkilöön on lisätty CoNLL-U-formaatti omaan lapsielementtiinsä.*

jalle vain kerran ja syntynyt *CoNLL-U* -formaatin merkkijono tallennetaan uudeksi lapsielementiksi myöhemmin analysoitavaan XML-tiedostoon kuten on havainnollistettu listauksessa 9.

Analysoitavien datatiedostojen rakenne siis tukee uusien tekstin esitystapojen lisäämistä myöhemmin dokumenttiin lapsielementtien avulla edellyttäen, että uusi formaatti on tekstimuotoista. Lapsielementit voidaan lisätä dokumenttiin joko erillisellä XML:ää käsittelevällä työkalulla tai kuten *CoNLL-U*:n tapauksessa kutsumalla erillistä *Kairan* komentoa, joka lukee aikaisemman XML-tiedoston raakatekstin ja tuottaa siitä *Finnish-dep-parserilla* uuden formaatin XML-tiedostoon liitettäväksi.

Kairan lukiessa XML-tiedostoa ajetaan tiedoston jokaiselle henkilöä vastaavalle elementille muunnosprosessi funktiolla *convert_xml_to_dict*, jossa henkilön data muutetaan Pythonin dict-olioksi, jota liukuhihnan louhintakomponentit myöhemmin käyttävät syötteenään. Oletuksena funktio poimii XML-elementin kaikki attribuutit (listauksessa 9 henkilön nimi ja arvioitu sivunumero) sekä lapsielementit *RAW* ja *CONLLU*. *CoNLL-U* merkkijono muunnetaan samalla Pythonilla käsiteltäväksi tietorakenteeksi käyttäen tarkoitukseen tarkoitettua kirjastoa.

CoNLL-U tuki päätettiin sisällyttää *Kairan* kehykseen, sillä se havaittiin hyödylliseksi menetelmäksi suomenkielisen tekstin analysoinnissa ja on siten käytettävissä kaikissa *Kairalla* todennäköisesti käsiteltävissä kirjasarjoissa. Tästä huolimatta arkkitehtuuriin mahdollistettiin laajennuspiste kirjasarjakohtaisesti. Uusia tekstiformaatteja lisätessä sovelluskehittäjä voi julistaa räätälöidyn version funktiosta *con-*

vert_xml_to_dict kirjasarjapluginin manifestitiedostossa. Tällöin kehyksen oletustoiminnallisuuden sijaan *Kaira* käynnistyessään ajaa määritellyn kirjasarjakohtaisen muunnosfunktion. Tämä mahdollistaa periaatteessa mielivaltaisten tekstipohjaisten formaattien välittämisen XML-tiedostosta louhintakomponenttien käytettäväksi.

Louhintakomponenteille välitetyn datan hyödyntäminen on niiden toteutuksesta riippuvaista. Louhintakomponentti saattaa käyttää yhtä tai useampaa dataformaattia syötteenään. Toisaalta komponentti ei välitä dataformaateista, joita se ei käytä. Näin ollen uusien dataformaattien lisääminen ei vaadi muutoksia liukuhihnan aikaisempaan toteutukseen tai sen komponentteihin.

5. TOTEUTUKSEN ARVIOINTI

Tässä luvussa arvioidaan *Kairan* arkkitehtuuria. Sovelluskehyksen arkkitehtuuria arvioidaan muutamalla erilaisella tavalla. Skenaariopohjaisessa arvioinnissa arkkitehtuurin laatuominaisuuksia arvioidaan erilaisten todennäköisten skenaarioiden pohjalta. Kukin skenaario paljastaa arkkitehtuurin vahvuuksia ja heikkouksia ja siten mahdollistaa arkkitehtuurin laatuvaatimusten toteutumisen arvioinnin.

Hypoteettisten skenaarioiden jälkeen *Kairan* arkkitehtuuria arvioidaan lyhyiden tapausesimerkkien avulla. Tapausesimerkeissä on kuvattu todellisten suoritettujen arkkitehtuurin muokkausoperaatioiden kulku. Näiden esimerkkien pohjalta on kuvattu kokemuksia ja arkkitehtuurin käytännön toimivuudesta heränneitä ajatuksia. Osaltaan tapausesimerkit myös peilaavat skenaarioiden paikkansapitävyyttä todellisuuteen nähden.

Nykytilanteen arvioinnin jälkeen arvioidaan arkkitehtuurin suunnittelu- ja toteutusprosessia sekä arkkitehtuurin tulevaisuuden potentiaalia tutkimusprojektin kontekstissa. Nykytilanteen ja tulevaisuuden kannalta arvioinnissa keskitytään erityisesti *Kairan* laajennettavuuteen, sillä sen todettiin suunnitteluvaiheessa olevan tärkein laatuominaisuus tutkimusprojektin onnistumisen kannalta.

5.1 Laajennettavuusskenaariot

Arkkitehtuurin laatua ja onnistumista voidaan arvioida skenaarioiden kautta, joiden uskotaan ilmentävän vaatimuksia, joihin arkkitehtuurin on pystyttävä tulevaisuudessa vastaamaan. *Kairan* tapauksessa oleellisimmat skenaariot liittyvät sen laajennettavuuteen. Laajennettavuus ja sovelluskehyksen soveltaminen uusiin kirjasarjoihin olivat alkuperäisiä sovelluskehyksen suunnittelun lähtökohtia.

Kairan sovelluskehyksen tärkein yleisluontoinen laatuvaatimus on tarjota joustavat laajennettavuusmahdollisuudet jatkokehitystä varten, sillä tutkimusprojekti edetessään tulee vaatimaan useita tietolähteitä (kirjasarjoja), joiden louhinta vaatii ohjelmistokehitystä. Tutkimusprojektin rajalliset resurssit huomioon ottaen tehokas ja nopea ohjelmistokehitys on avain hankkeen onnistumiseen. Niinpä oleellisinta on arvioida juuri laajennettavuusaspektia sovelluskehyksestä. Vastaavasti esimerkiksi suorituskykyyn ei suunnittelussa juurikaan paneuduttu pienehköjen datamäärien

vuoksi. Lisäksi esimerkiksi käyttäjäkokemuksen optimointi oli tutkimuskäyttöön tarkoitetun komentorivisovelluksen toteutuksessa toissijaista.

Tässä luvussa esitellään tärkeimmät laajennettavuuteen liittyvät skenaariot. Skenaarioiden tuottamisessa sovellettiin luvussa 2 esiteltyä Bachmanin ja Kleinin mallia. Skenaarioiden osien tunnistamisen jälkeen niistä muodostettiin alkuperäistä mallia vapaamuotoisemmat ja kattavammat kuvaukset. Skenaarioissa kuvataan niiden perustilanne, eli mitä uusia vaatimuksia sovelluskehykselle ilmenee, keneen luontaiseen henkilöön skenaario vaikuttaa ja miten arkkitehtuuri kykenee vastaamaan skenaarion tilanteeseen. Kuvauksessa käydään myös korkealla tasolla läpi toimenpiteet, joita skenaario tulee vaatimaan sovelluskehittäjältä tai muulta skenaarioon liittyvältä luontaiselta henkilöltä.

Kairan laajennettavuusskenaariot voidaan karkeasti jakaa kolmeen ryhmään:

1. kirjasarjatoteutuksen laajentaminen
2. uuden kirjasarjan toteuttaminen
3. uusien lähdedataformaattien lisääminen

Kustakin ryhmästä on valittu yksi skenaario esiteltäväksi, joka kuvaa ryhmälle ominaiset piirteet, vaatimukset ja tarvittavat toimenpiteet. Ryhmien vaatimat toimenpiteet ovat kuitenkin keskenään erilaisia ja vuorovaikuttavat arkkitehtuurin eri osien ja laajennosmekanismien kanssa. Siten skenaariot ovat laadullisesti toisistaan erilaisia ja ilmentävät erilaisia arkkitehtuurin laatuominaisuuksia. Kolmen laajennettavuusskenaariokategorian lisäksi käsitellään myös lyhyesti sovelluskehyksen potentiaalia täysin erilaisen tekstin louhintaan tutkimusprojektin ulkopuolella.

5.1.1 Kirjasarjan laajentaminen louhintakomponenteilla

Useimmat tutkimusprojektin tarvitsemat kirjasarjat sisältävät paljon dataa, jota varten on kehitettävä useita erilaisia louhintakomponentteja. Siten yleisin laajennettavuusskenaario voidaan kuvata seuraavasti:

Kirjasarjan liukuhihnalle on lisättävä louhintakomponentti, joka irrottaa tekstistä uuden datan kappaleen ja tallentaa sen tulosjoukkoon. Komponentti saattaa olla itsenäinen tai riippuvainen muista liukuhihnalla olevista louhintakomponenteista.

Tämä skenaario on yleinen, sillä jokaisen kirjasarjan toteutus vaatii useiden louhintakomponenttien kirjoittamista ja on siten arkkitehtuurin perusrakenneosa, jonka kanssa sovelluskehittäjä työskentelee. *Kairan* perusarkkitehtuuri rakentuukin louhintakomponenttien ja niiden muodostaman liukuhihnan ympärille ja on siten suunniteltu vastaamaan tähän nimenomaiseen skenaarioon.

Uuden louhintakomponentin lisääminen vaatii sovelluskehittäjältä ymmärryksen *Kairan* liukuhihna-arkkitehtuurin rakenteesta, komponenttien toteuttamisesta ja niiden konfiguroinnista. Uuden komponentin kirjoittaminen aloitetaan lisäämällä koodikantaan uusi luokka, joka perii *BaseExtractor* komponentin toteutuksen. Uuden komponentin tulee toteuttaa vähintään *_extract*-metodi, ja sen tulee kunnioittaa *BaseExtractor*-luokan määrittelemää esi- ja jälkiehtorajapintasopimusta. Luokan luominen vaatii siis perehtymistä sovelluskehyksen tarjoaman *BaseExtractor*-luokan rajapintaan sekä sen toimintaan liukuhihnan osana.

Uuden luokan lisäämisen jälkeen sovelluskehittäjä voi kirjoittaa luokkaan sisäisen toteutuksen, joka käsittelee sille välitettyä tekstiä ja palauttaa louhimansa datan *_extract*-metodin paluuarvona kehyksen edellyttämässä formaatissa. *Kaira* ei määrittele tarkemmin louhintakomponentin sisäisen toiminnallisuuden toteutustapaa. Tärkeintä on huolehtia komponentin rajapinnan toteuttamisesta kehyksen kanssa yhteensopivaksi. Poikkeus sääntöön on myös olettama komponentin tilattomuudesta ajon aikana.

Komponentin kirjoittamisen yhteydessä sovelluskehittäjän on suotavaa kirjoittaa komponentille yksikkötestit oikean toiminnallisuuden varmistamiseksi. Testit voidaan kirjoittaa yleensä helposti komponenttikohtaisiksi, sillä komponenttien tilattomuuden vuoksi testit voidaan rakentaa ilman laajaa lisäriippuvuuksien simulointia.

Komponentin määrittelyn jälkeen sovelluskehittäjän tulee konfiguroida komponentti osaksi ajettavaa liukuhihnaa lisäämällä sille tietue kirjasarjan omaan *pipeline_config.yaml* -tiedostoon. Tietueen lisääminen vaatii ymmärrystä konfigurointiformaatin rakenteesta ja sen tukemista asetuksista. Yksinkertaisimmillaan sovelluskehittäjä voi kuitenkin lisätä komponentin haluamaansa kohtaan konfiguraatiota ja määrittää sille mahdolliset asetukset. Konfiguraation jälkeen komponentti on osa liukuhihnaa ja seuraavan ajon yhteydessä *Kaira* suorittaa siihen lisätyn ohjelmakoodin osana liukuhihnaa.

Skenaarion toimenpiteet eivät vaadi sovelluskehyskoodin muokkaamista millään tavalla. Muutokset rajoittuvat yhteen louhintakomponenttiin sekä sen konfiguraatioon. Arkkitehtuuri siis rajaa muutokset pieneen osaan sovellusta ja siten tukee uusien komponenttien lisäämistä järjestelmään suhteellisen vähällä vaivalla.

Louhintakomponenttien lisäämisen vaatima työ kuitenkin vaihtelee komponentista riippuen. Edellä kuvattu yksinkertainen muista riippumaton komponentti on suhteellisen yksinkertaista lisätä järjestelmään. Sen sijaan muiden komponenttien tulosjoukosta riippuvien uusien komponenttien lisääminen järjestelmään on hieman mutkikkaampaa. Tämä edellyttää riippuvuussuhteiden määrittämisen konfiguraatiotiedostoon oikein, jotta komponentin tarvitsema data voidaan välittää sille oikein. Tämä vaatii syvällisempää ymmärrystä paitsi sovelluskehyksen toiminnasta

myös ratkaistavasta ongelmasta itsestään. Riippuvuuksien määrittely myös vaatii monimutkaisempaa testien määrittelyä, sillä testejä varten kehittäjän on simuloitava riippuvuuksien injektoiminen testattavaan komponenttiin. *Kaira* kuitenkin tarjoaa näihin tarkoituksiin yksinkertaisia työkaluja, jotka helpottavat testien käytännön kirjoittamista.

Vaikka riippuvuussuhteita sisältävien louhintakomponenttien lisääminen kehykseen on monimutkaisempi prosessi, sovelluskehys kuitenkin erottaa edelleen komponentit toisistaan erillisiksi yksiköiksi, jolloin ne ovat edelleen erikseen testattavissa ja uudelleenkäytettävissä. Tarkempi käytännön tapausesimerkki uusien komponenttien lisäämisestä esitellään aliluvussa 5.2.

5.1.2 Uuden kirjasarjan tuen lisääminen

Tutkimusprojektin edetessä tullaan tarvitsemaan dataa useista eri matrikkelikirjasarjoista. Kukin erillinen kirjasarja vaatii sille ominaisen liukuhihnan, joka koostuu louhintakomponenteista. Tämä laajennettavuusskenaario voidaan ilmaista seuraavasti:

Kairaan tarvitaan tuki uudelle kirjasarjalle, jonka sisältö poikkeaa suurelta osin aikaisemmin toteutetuista kirjasarjoista. Osa kirjasarjan datasta on samankaltaista muiden kirjasarjojen kanssa, mutta suuri osa on ainutlaatuista tai rakenteeltaan muista kirjasarjoista poikkeavaa.

Koska uuden kirjasarjan datan laatu ja rakenne poikkeavat muista kirjasarjoista, ei ole kannattavaa yrittää rakentaa sille tukea toiselle kirjasarjalle tarkoitettuun liukuhihnakonfiguraatioon. Olemassa olevan kirjasarjakonfiguraation sijaan sovelluskehittäjän on lisättävä tuki kokonaan uudelle kirjasarjalle.

Kirjasarjan lisääminen vaatii ensimmäiseksi *chunking*-operaation toteuttamisen. Tämä sovelluksen osa käytännössä lukee kirjasarjan lähdeformaatin, esimerkiksi HTML-tiedoston, ja muuntaa sen *Kairalle* yhteensopivaan XML-formaattiin.

Seuraavaksi sovelluskehittäjän on luotava uuden kirjasarjan vaatimat peruskonfiguraatiotiedostot *manifest.yaml* ja *pipeline_ config.yaml* ja sijoitettava nämä tiedostot uuden hakemiston alle polkuun *extractors/bookseries*. Näiden tiedostojen pohjalta *Kairan* plugin-arkkitehtuurimekanismi kykenee tunnistamaan tuen uudelle kirjasarjalle ja mahdollistaa sen ajamisen komentoriviltä kirjasarjakohtaisella komentoriviparametrilla.

Tämä skenaario edustaa korkeamman tason laajennettavuutta yksittäisten louhintakomponenttien toteuttamiseen verrattuna. Skenaario hyödyntää pluginarkkitehtuuria eriyttämään eri kirjasarjojen toteutuksen toisistaan. Muutokset yhden kirja-

sarjan toteutukseen eivät vaikuta muiden kirjasarjojen toimintaan. Skenaario vaatii sovelluskehittäjältä perusymmärryksen *Kairan* plugin-arkkitehtuurista ja sen vaatimasta konfiguraatiosta.

Plugin-konfiguraation jälkeen sovelluskehittäjä voi kehittää yksittäisiä louhintakomponentteja uuteen kirjasarjaan normaaliin tapaan. Poikkeus tähän on mahdollisuus jakaa toiminnallisuutta eri kirjasarjojen välillä. Usein eri kirjasarjat sisältävät samankaltaista dataa, jonka irrottamiseen voidaan käyttää samankaltaista logiikkaa. Tällöin eri kirjasarjat voivat periyttää osan komponenteistaan yhteisistä louhintakomponenteista. Tällöin kukin erillinen kirjasarja kuitenkin erikoistaa yhteisen toteutuksen perustoiminnallisuuden omaan käyttöönsä sopivaksi vaikuttamatta toistensa toteutukseen.

Skenaarion suurin muuttuja on kirjasarjan lähdedatan käsittelyyn vaadittavan *chunkerin* toteutus. Riippuen lähdemateriaalista sen toteuttaminen voi vaatia vaihtelevan määrän työtä ja testausta. Sitä lukuunottamatta uusien kirjasarjojen lisääminen sovelluskehykseen ei juuri eroa kirjasarjakohtaisesti ja se voidaan toteuttaa aina samalla suoraviivaisella periaatteella.

5.1.3 Uuden lähdedataformaatin lisääminen

Tavallisesti *Kaira* lukee louhittavan tekstin XML-tiedostosta, joka sisältää raakatekstin ja mahdollisesti esikäsitellyn formaatin raakatekstistä. Riippuen kirjasarjan tarvitsemista algoritmeista saattaa olla tarpeen esikäsitellä teksti algoritmille sopivaan muotoon:

Kirjasarjan louhintakomponentit tarvitsevat ennakkoon käsiteltyä formaattia raakatekstistä. Raakatekstiin on esimerkiksi täydennettävä sanaluokat tai lauseen sanojen välisiä suhteita, joita ei kannata rakentaa tekstin louhinnan yhteydessä.

Skenaario on huomattavasti aiempia esiteltyjä skenaarioita harvinaisempi, mutta sen kuvaama tilanne on tarpeen esimerkiksi edistyneen luonnollisen kielen käsittelyn yhteydessä. Skenaario on realisoitunut esimerkiksi *ConNLL-U*-formaatin lisäämisessä osaksi mahdollista *Kairan* syötedataa.

Skenaarion ensimmäinen vaihe on uuden lähdedataformaatin yhdistäminen osaksi *Kairan* lukemaa XML-formaattia. Käytännössä formaattiin on lisättävä uusi XML-tagi, joka sisältää uuden dataformaatin, jota louhinnassa halutaan hyödyntää. Datan lisääminen syötetiedostoon voidaan tehdä osana *Kairan* chunking-prosessia, jossa kyseinen tiedosto luodaan. Vaihtoehtoisesti data voidaan lisätä jälkeenpäin millä tahansa XML-tiedostojen muokkaamiseen kykenevällä ohjelmalla tai ohjelmointikielellä.

Seuraavaksi *Kairan* sovelluskehystä on laajennettava lukemaan uutta dataformaattia kun tiedosto ladataan *Kairan* käsiteltäväksi. Tähän tehtävään sovelluskehys tarjoaa valmiin kirjasarjakohtaisen mekanismin. Kirjasarja voi manifestitiedostossaan määrittää ominaisuuden *xml_to_dict*, joka sisältää polun Python-moduuliin, jossa sijaitsee funktio *convert_xml_to_dict*. Ohjelmoija voi määritellä räätälöidyn toteutuksen tähän funktioon, jonka *Kaira* ajaa tiedostonluvun yhteydessä oletustoteutuksensa sijaan. Funktion tehtävänä on lukea XML-tiedoston elementit yksi kerrallaan ja muuntaa ne Python-objektiksi. Funktion sisäistä toteutusta muuttamalla voidaan näin kirjoittaa tuki uusille dataformaateille, joita louhintakomponentit voivat louhintaprosesseissaan hyödyntää.

Kuvailtu laajentamismekanismi dataformaateille on yksinkertainen toteuttaa, mutta on myös osittain rajoittunut. Mekanismi olettaa uuden dataformaatin olevan tekstipohjaista dataa, joka voidaan helposti liittää osaksi XML-tiedostoa, joten esimerkiksi binäärimuotoinen data ei suoraan sovi laajennusrajapintaan. Toisaalta kuitenkin toteutettava räätälöity versio funktiosta voi sisältää mitä tahansa Python koodia, joka esimerkiksi lataa dataa ulkopuolisista lähteistä tai verkosta ajon aikana ja muuntaa kyseisen datan Python-objektiin sopivaan muotoon. Tällaisille prosesseille ei kuitenkaan ole sovelluskehyksessä valmista tukea, joten niiden toteuttaminen jää kokonaan sovelluskehittäjän vastuulle. Olemassa olevan laajennosrajapinnan kuitenkin tulisi vastata useimpiin mahdollisiin uusien dataformaattien tarpeisiin.

5.1.4 Matrikkeliformaatista poikkeavan tekstin louhinta

Kaira suunniteltiin ensisijaisesti tutkimusprojektin tarvitsemia matrikkelikirjasarjoja varten jättäen huomiotta muut mahdolliset tekstinlouhinnan sovelluskohteet. Tämä sovelluskohteiden rajaaminen matrikkelikirjasarjoihin ohjasi suunnittelupäätöksiin, jotka todennäköisesti rajaavat *Kairan* käytettävyyttä muissa tekstinlouhintatehtävissä.

Sovelluskehys olettaa louhittavan tekstimassan koostuvan lyhyehköistä toisistaan erillisistä tekstikappaleista, sillä matrikkelikirjoissa eri henkilöiden tiedot on jaettu erillisiin tekstikappaleisiin. *Kairan* liukuhihna käsittelee kerrallaan yhden tekstikappaleen ja koostaa siitä tulosjoukon. Kirjan kaikista tekstikappaleista koostetaan samanlaiset tietojoukot, jotka lopulta tallennetaan listaksi. Tämä olettama todennäköisesti ei toimi erityisen hyvin esimerkiksi seuraavassa skenaariossa:

Sovelluskehystä halutaan käyttää irrottamaan skannatusta sanomalehdestä sanomalehtiartikkeleissa mainittuja tietoja.

Sanomalehden artikkelit ovat jaettavissa erillisiksi yksittäisiksi tekstikappaleiksi, jotka ovat periaatteessa *Kairan* käsiteltävissä. Ongelmaksi kuitenkin muodostuu ar-

tikkelien keskinäinen erilaisuus. Eri artikkelit todennäköisesti tarvitsevat erilaisia tiedonlouhinta-algoritmeja. Käytännössä tämä vaatisi eri artikkeleille erilaiset liukuhihnamäärittelyt, jotka *Kairan* arkkitehtuurissa on sidottu kirjakohtaisiksi. Tämän skenaarion tukeminen siis vaatisi merkittäviä muutoksia *Kairan* perusarkkitehtuuriin, jossa käytännössä kirjasarjalle pitäisi pystyä määrittämään useita liukuhihnoja, jotka ajetaan juuri tietyntyyppisille tekstikappaleille.

Seuraava skenaario ilmaisee toisen *Kairan* arkkitehtuurin rajoitteen:

Kairaa halutaan hyödyntää tiedon louhintaan romaanista tai muusta jatkuvaa kerrontaa sisältävästä tekstistä.

Kaira suunniteltiin lyhyille, keskenään samankaltaisille tekstikappaleille. Esimerkiksi romaanin teksti todennäköisesti on jatkuvasti muuttuvaa, eikä sisällä systemaattisesti samankaltaisina toistuvia tekstirakenteita. Lisäksi tekstin merkitysten ymmärtäminen saattaa vaatia tekstin suurempien rakenteiden ymmärtämistä. Tällaista tekstiä olisi todennäköisesti vaikea sovittaa *Kairan* liukuhihnamalliin, jossa jokainen komponentti irrottaa tekstistä tietyn datan kappaleen.

5.2 Tapaus: Suomen rintamamiehet -kirjasarja

Suomen Rintamamiehet 1939-1945 on matrikkelikirjasarja, johon on koottu henkilötietoja sotilaista, jotka osallistuivat talvi- ja jatkosotaan. Kirjat luettelevat rintamamiehistä muun muassa heidän siviiliammattinsa, puolison, lapset, sodanaikaiset sijoitukset ja sotilasarvon. Rakenteeltaan teksti vastaa hyvin formaattia, johon *Kairaa* voidaan soveltaa, joten kirjasarjaa päätettiin käyttää testikohteena *Kairan* soveltamiselle uuteen kirjasarjaan. Tarkoituksena on tutkia sovelluskehyksen toimivuutta ja soveltuvuutta uuteen kirjasarjaan sovelluskehittäjän näkökulmasta pyrkien myös tunnistamaan mahdolliset puutteet ja parannuskohteet. Tässä aliluvussa kuvataan lyhyesti käytännön kokemuksia ja havaintoja uuden kirjasarjan lisäämisestä kehykseen.

Kirjasarjan louhintalogiikan kehitys aloitettiin lisäämällä uusi hakemisto *bookseries*-hakemiston alle, jonne määritettiin kirjasarjalle ominaiset *manifest.yaml* ja *config.yaml*. Ensimmäisenä kehitettiin *ABBYY Finereaderin* tuottaman HTML-tiedoston lukeva ja käsittelevä ohjelmistokomponentti, joka tuottaa analysoitavissa olevan XML-tiedoston. Komponentille määriteltiin *convert_html_file_to_xml* -funktio ja komponentin polku määritettiin kirjasarjan manifesti-tiedostoon. Näiden toimenpiteiden jälkeen uuden kirjasarjan muuntaminen XML-formaattiin oli mahdollista *Kairan* plugin-arkkitehtuurin ansiosta, joka käynnistyksen yhteydessä rekisteröi uuden kirjasarjan sille määritellyn manifestin pohjalta.

Louhintakomponentin kirjoittaminen

Kun XML-tiedoston rakentamiskomponentti oli toteutettu, voitiin aloittaa varsinaisen tiedonlouhintalogiikan rakentaminen. Ensimmäisenä toteutetut komponentit vastasivat sotilaan nimitietojen, syntymäpäivän ja syntymäpaikan irrottamisesta tekstistä. Nimen louhintaan voitiin ottaa pohjaksi muulle kirjasarjalle toteutettu louhintakomponentti, jonka toteutus vaihdettiin uutta käyttötarkoitusta varten. Louhintakomponentti periytettiin sovelluskehyksen *BaseExtractor*-luokasta ja sille kirjoitettiin tarvittava kirjasarjakohtainen toteutus. Syntymäajan ja -paikan irrottamiseen tekstistä käytettiin kehyksen tarjoamaa yleistä komponenttia, josta periytettiin sotilaskirjasarjaan erikoistettu uusi komponentti.

Komponentit lisättiin kirjasarjan *config.yaml*-tiedostoon, jonka pohjalta *Kaira* automaattisesti rakentaa kirjasarjakohtaisen liukuhihnan. Konfiguraation lisäämisen jälkeen kirjasarjan louhintaprosessi oli heti ajettavissa uudelle kirjasarjalle ja sovelluskehys tuotti onnistuneesti JSON-dokumentin, joka sisälsi irrotetut tiedot kirjasarjan kaikille ensisijaisille henkilöille.

Ensimmäisten louhintakomponenttien toteutuksen yhteydessä huomattiin kuitenkin parantamisen varaa testikehyksen käytössä. Erityisesti louhintakomponenttien alustaminen yksikkötestejä varten vaatii sovelluskehyksen testityökalujen tuntemista ja ymmärtämistä. Esimerkiksi louhintakomponentin alustamiseen vaaditaan sovelluskehyksen tarjoaman *th (test helper)* -luokan tarjoamaa *setup_extractor*-metodia. Lisäksi periyttäminen ja uusien komponenttien rakentaminen vaativat sovelluskehittäjältä hyvää *Kairan* luokkien rajapintojen ymmärrystä. Nämä ongelmat ovat kuitenkin ratkaistavissa selkeällä dokumentaatiolla ja esimerkkikoodilla. Lisäksi tulevaisuudessa voidaan testausta avustavia luokkia ja työkaluja kehittää tarvittaessa helppokäyttöisemmiksi.

5.3 Arkkitehtuurin suunnittelu kehitysprosessin aikana

Arkkitehtuurin suunnitteluun ja sen toteuttamiseen vaikutti oleellisesti olemassa oleva prototyyppi. Kehitys aloitettiin pikaisella arviolla prototyypin ongelmakohdista ja vahvuuksista, jolloin tultiin lopputulokseen käyttää sitä pohjana jatkokehitykselle täydellisen uudelleenkirjoittamisen sijaan. Valinta perustettiin prototyypin sisältämään louhintakomponenttien logiikkaan, jota arvioitiin voitavan hyödyntää jatkossa.

Kehitysprosessi aloitettiin prototyypin arkkitehtuurin ongelmakohtien tunnistamisella, sekä poistamalla suurin osa jatkon kannalta hyödyttömästä koodista. Samalla

projektiin kirjoitettiin tärkeimpien komponenttien yhteyteen yksikkötestejä helpottamaan tulevaa refaktorointiprosessia.

Suuri osa kehityksen alkuvaiheesta käytettiin iteratiiviseen refaktorointiin. Hyvän domain-tuntemuksen ja pienehkön ja tutun koodikannan ansiosta työ voitiin aloittaa tunnistamalla aluksi ensimmäiset tarvittavat muutokset arkkitehtuuriin. Esimerkiksi arkkitehtuuriin päätettiin luoda eksplisiittinen tietovuoarkkitehtuuri. Kun tarvittava muutos arkkitehtuurista oli tunnistettu, suunniteltiin tarvittavat refaktorointivaiheet muutosten saavuttamiseksi. Muutoksia suunniteltiin ajoittain useita eteenpäin, sillä toivotut arkkitehtuurimuutokset saattoivat olla riippuvaisia aikaisemmista muutoksista.

Jokaisen muutoksen yhteydessä kirjoitettiin muutosta vastaavat yksikkötestit, joilla varmistettiin refaktoroinnin edetessä sovelluksen toimivuus. Yksikkötestien kertyessä refaktorointi nopeutui ja tarvittavien muutosten tekeminen helpottui. Tämä käytäntö rohkaisi jatkossa arkkitehtuurin jatkuvaan uudelleen arviointiin ja muokkaamiseen tarpeita vastaavaksi evolutiivisella tavalla. Mahdolliset epäonnistuneet arkkitehtuuripäätökset refaktoroitiin myöhemmin pois, kun niiden ongelmat havaittiin.

Tämä evolutiivinen arkkitehtuurin suunnittelu ei olisi ollut mahdollista ilman yksikkötestien kurinalaista kirjoittamista kunkin refaktorointisyklin aikana. Tehtävässä myös auttoi koodikannan suhteellisen pieni koko. Käytetty menetelmä mahdollisti rohkean refaktoroinnin ja siten rakenteen ketterän kehittämisen. Menetelmässä oli myös mahdollista viivästää arkkitehtuurillisia suunnittelupäätöksiä myöhempään ajankohtaan. Suunnittelupäätösten viivästämisellä on etuna domaintietämyksen kasvaminen projektin edetessä, jolloin on helpompi arvioida suunnittelupäätösten toteutuskelpoisuutta ja hyödyllisyyttä [12]. Viivästämisellä voidaan myös välttää turhien ominaisuuksien kehittäminen ennenaikaisesti. Tämä opittiin myös projektin aikana, sillä kerran arkkitehtuurista päädyttiin poistamaan aiemmin kehitetty ominaisuus, jolle ei lopulta nähty hyödyllistä käyttötarkoitusta.

Evolutiivinen arkkitehtuurisuunnittelu todettiin onnistuneeksi valinnaksi projektin aikana uusien vaatimusten noustessa esiin louhittavan datan suhteen. Sovelluksen louhima data oli tutkimusprojektin käytettävissä alusta saakka koko kehitysprosessin ajan, joten parannuksia ohjelmistoon kehitettiin käyttäjiltä saadun palautteen ja toiveiden perusteella.

Evolutiivista arkkitehtuurisuunnittelua oli erityisen helppo soveltaa ohjelmiston pienen koodikannan vuoksi, joten samaa menetelmää olisi mahdollisesti hankalampaa käyttää isommissa projekteissa ainakaan ilman kattavampaa valmista automaattitestausta. Lisäksi evolutiivinen arkkitehtuurisuunnittelu vaatii jatkuvaa tarpeen mu-

kaan suoritettavaa refaktorointia ja suunnittelupäätösten uudelleenarviointia. Muutoin menetelmä helposti johtaa arkkitehtuuriin, joka koostuu *ad hoc* suunnittelupäätöksistä ilman suurempaa kokonaisrakennetta. Menetelmä myös vaatii tulevien muutosten huomiointia, sillä tehdyt muutokset arkkitehtuuriin vaikuttavat todennäköisesti myös myöhempiin muutoksiin. [12]

5.4 Jatkokehitys

Kairan arkkitehtuuri suunniteltiin laajentamista ja siten jatkokehitystä silmällä pitäen. Tutkimusprojektin edetessä tarpeet tarkemmalle tiedonlouhinnalle kasvavat. Projektin vaatiessa tutkimuskysymyksiinsä useita aineistoja syntyy tarve myös uusien kirjasarjojen tukemiselle.

Kairan soveltamisen lisäksi uusiin kirjasarjoihin voi syntyä tarvetta perusteellisemmille kehyksen rakenteeseen vaikuttaville muutoksille. Tällainen on esimerkiksi luonnollisen kielen käsittelyyn *(NLP, natural language processing)* liittyvä, jo aiemmin mainittu *CoNLL-U*-formaatti. Formaattiin perustuen voidaan kehittää uudenlaisia, monimutkaisempia ja tehokkaampia luonnollisen kielen louhintaan tarkoitettuja algoritmeja. *Kairaan* rakennettiin jo aiemmin tuki formaatin lukemiseen XML-tiedostosta. Tulevaisuudessa, kun *CoNLL-U* otetaan käyttöön useammissa louhintakomponenteissa, saattaa esiin nousta uusia tarpeita, joita sovelluskehyksen tulisi tukea auttaakseen ohjelmoijaa kehitystyössään. Esimerkki tällaisesta mahdollisuudesta on matalan tason NLP-toimintojen abstrahoiminen helppokäyttöisemmän rajapinnan taakse. Työtä tällaisen työkalun suunnitteluun tai sen tarpeellisuuden kartoitukseen ei kuitenkaan ole vielä tehty.

Sovelluskehyksen kehityksessä ei toistaiseksi ole kiinnitetty erityistä huomiota suorituskyvyn optimointiin. Toistaiseksi suorituskyky on ollut tarpeeksi nopeaa analysoitavien datajoukkojen käsittelyyn, sillä analyysit ajetaan tyypillisesti harvakseltaan eräajoina tietokantaan. Tästä huolimatta suorituskyvyn voidaan katsoa olevan sovelluskehyksen käyttökelpoisuutta rajoittava tekijä.

Todennäköisesti merkittävin suorituskykyä parantava tekniikka olisi rinnakkaistaa liukuhihnan louhintakomponenttien suoritus. Komponentit ovat itsessään tilattomia ja vain lisäävät dataa tulosjoukkoon. Periaatteessa siis teksti voitaisiin syöttää useille rinnakkain ajettaville louhintakomponenteille, joiden tulokset lopulta koottaisiin yhteen yhdeksi tulosjoukoksi. Tämä rinnakkaistaminen sisältää kuitenkin haasteita. Louhintakomponenttien ollessa riippuvaisia toisistaan täytyisi myös rinnakkaistaminen koordinoida oikein, jotta sovellus välttyisi esimerkiksi tuottaja-kuluttaja-ongelmalta [2], jossa osa louhintakomponenteista joutuu pitkään odottamaan hitaampien louhintakomponenttien prosessien valmistumista. Riippuen siitä, kuinka

louhintakomponentit on jaettu rinnakkaisiin prosesseihin, tulisi kehyksen myös huolehtia tehokkaasta viestin välityksestä prosessien välillä.

Rinnakkaistamisen lisäksi isojen datajoukkojen käsittelyssä louhintalogiikan toteuttaminen pelkällä Python-kielellä ei todennäköisesti ole kannattavaa. Suorituskykyä voitaisiin parantaa esimerkiksi profiloinnilla tunnistamalla sovelluksen hitaimmat pullonkaulat ja toteuttamalla nämä louhinta-algoritmit esimerkiksi C-kielellä käyttäen Pythonin C-laajennosominaisuutta [9].

Nämä suorituskykyä parantavat ratkaisut eivät sinänsä ole poikkeuksellisia tai erityisen vaikeita toteuttaa osaksi *Kairan* arkkitehtuuria. Tutkimusprojektin tavoitteet ja prioriteetit ovat kuitenkin toistaiseksi ohjanneet kehitystyötä keskittymään muihin laadullisiin ominaisuuksiin.

6. YHTEENVETO

Kaira on *Learning from our Past* -tutkimusprojektiin kehitetty sovelluskehys, jota käytetään tiedon louhintaan digitoiduista matrikkelikirjasarjoista. Kehys suunniteltiin erityisesti nopeaa laajentamista varten, jotta kerran kehitettyä tiedonlouhintalogiikkaa voitaisiin hyödyntää useisiin eri kirjasarjoihin. Lisäksi suunnittelussa huomioitiin olemassa olevien kirjasarjojen louhintalogiikan laajennettavuustarpeet.

Sovelluskehyksen arkkitehtuurin tärkeimmät ominaisuudet laajennettavuuden suhteen ovat tietovuoarkkitehtuuri ja plugin-arkkitehtuuri. Tietovuoarkkitehtuurille tyypillinen liukuhihnarakenne toimii kirjasarjan louhintalogiikkaa hallinnoivana rakenteena ja mahdollistaa uusien louhinta-algoritmien lisäämisen yksittäisinä liukuhihnakomponentteina. Plugin-arkkitehtuuri rakennettiin ratkaisuksi uusien kirjasarjojen lisäämiseksi sovelluksella analysoitavaksi.

Arkkitehtuuri suunniteltiin kiinnittäen erityistä huomiota *taktiikoihin*, jotka tukisivat laajennettavuutta ja minimoisivat esimerkiksi *värevaikutukset* sovellusta muokattaessa. Sovelluskehyksen kehitystyössä ja suunnittelussa hyödynnettiin evolutiivista arkkitehtuurisuunnittelua, jonka keskiössä olivat kattava automaattitestaus ja toistuva refaktorointi. Tämä kehitysprosessi ja käytetyt arkkitehtuuriratkaisut mahdollistivat nopean reagoinnin uusiin käyttötarpeisiin ja sovelluskehyksen jatkuvan parantamisen tutkimusprojektin edetessä.

Luvussa 5 suoritetun analyysin perusteella valittujen arkkitehtuuriratkaisujen todettiin vastaavan tutkimusprojektin vaatimuksia. Arkkitehtuuriratkaisut tukevat yleisimpiä tutkimusprojektin kontekstissa ilmeneviä skenaarioita. Yleisimmät skenaariot sisältävät sovelluksen laajentamisen uusilla louhintakomponenteilla ja kirjasarjoilla. Toisaalta käytetyt ratkaisut myös rajoittavat sovelluksen käyttöä tutkimusprojektin kontekstin ulkopuolella, kun analysoitava teksti poikkeaa matrikkelikirjasarjoille tyypillisestä formaatista tai sovelluskohde vaatii korkeaa suorituskykyä.

Kirjoitushetkellä *Kaira* tukee kolmen matrikkelikirjasarjan datan louhintaa. Lisäksi muutamia louhintakomponentteja on kirjoitettu neljännelle kirjasarjalle, jonka yhteydessä on hyödynnetty myös aikaisemmin kirjoitettuja louhintakomponentteja. Sovelluksen tuottamaa dataa käytetään aktiivisesti yhtenä lähdeaineistona *Learning from our Past* -tutkimusprojektissa. *Kairan* kehitystyötä jatketaan edelleen lisäten siihen jatkuvasti uusia tarvittavia ominaisuuksia tutkimusprojektin edetessä.

LÄHTEET

[1] ABBYY Finereader 14, verkkosivu, tietokoneohjelma. Saatavissa (viitattu 28.10.2018): https://www.abbyy.com/en-eu/finereader/

[2] R.H. Arpaci-Dusseau, A.C. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, Arpaci-Dusseau Books, 2015, 714 p.

[3] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley Professional, 2013, 640 p.

[4] M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowloer, J. Highsmith, A. Hunt, J. Kern, B. Marick, R.C. Martin, K. Schwaber, J. Sutherland, D. Thomas, Manifesto for Agile Software Development, agilemanifesto.org, verkkosivu, 2001. Saatavissa (viitattu 31.3.2018): http://www.agilemanifesto.org

[5] CoNLL-U Format, universaldependencies.org, verkkosivu, 2017. Saatavissa (viitattu 6.6.2018): http://universaldependencies.org/format.html

[6] Data Flow Architecture, tutorialspoint.com, verkkosivu. Saatavissa (viitattu 15.1.2018): https://www.tutorialspoint.com/software_architecture_design/data_flow_architecture.htm

[7] Understanding how Eclipse plug-ins work with OSGi, ibm.com, verkkosivu, 2006. Saatavissa (viitattu 4.6.2018): https://www.ibm.com/developerworks/library/os-ecl-osgi/index.html

[8] V.P. Eloranta, U. Heesch van, P. Avgeriou, N. Harrison, K. Koskimies, Lightweight Evaluation of Software Architecture Decisions, in: Mistrik, I., Bahsoon, R., Eeles, P., Roshandel, R., Stal, M. (eds.), Relating System Quality and Software Architecture, Morgan Kaufmann, Massachusetts, USA, 2014, pp. 157–179.

[9] Extending Python with C or C++, docs.python.org, verkkosivu, 2018. Saatavissa (viitattu 20.10.2018): https://docs.python.org/3.7/extending/extending.html

[10] M. Fowler, HarvestedFramework, MartinFowler.com, verkkosivu, 2003. Saatavissa (viitattu 10.12.2017): https://martinfowler.com/bliki/HarvestedFramework.html

[11] M. Fowler, Inversion of Control Containers and the Dependency Injection pattern, MartinFowler.com, verkkosivu, 2004. Saatavissa (viitattu 18.3.2018): https://martinfowler.com/articles/injection.html

[12] M. Fowler, IsDesignDead, MartinFowler.com, verkkosivu, 2004. Saatavissa (viitattu 27.6.2018): https://www.martinfowler.com/articles/designDead.html

[13] M. Fowler, InversionOfControl, MartinFowler.com, verkkosivu, 2005. Saatavissa (viitattu 8.1.2018): https://martinfowler.com/bliki/InversionOfControl.html

[14] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, E. Gamma, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 1999, 431 p.

[15] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995, 395 p.

[16] F. Ginter, J. Kanerva, S. Pyysalo, Finnish-dep-parser An Open Source dependency parsing pipeline for Finnish, Github, tietokoneohjelma, 2014. Saatavissa (viitattu 20.10.2018): http://turkunlp.github.io/Finnish-dep-parser/

[17] K. Koskimies, T. Mikkonen, Ohjelmistoarkkitehtuurit, Talentum Media, 2005, 250 s.

[18] J. Loehr, R. Lynch, J. Mappes, T. Salmi, J. Pettay, V. Lummaa, Newly Digitized Database Reveals the Lives and Families of Forced Migrants from Finnish Karelia, Finnish Yearbook of Population Research, Vol. 52, Iss. 19, 2017, pp. 9245–9252.

[19] M. Mattson, H. Grahn, F. Mårtensson, Software Architecture Evaluation Methods for Performance, Maintainability, Testability, and Portability, citeseerx.ist.psu.edu, verkkosivu. Saatavissa (viitattu 17.7.2018): http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.218.4705&rep=rep1&type=pdf

[20] A. Patidal, U. Suman, A Survey on Software Architecture Evaluation Methods, in: 2015 2nd International Conference on Computing for Sustainable Global Development, Ft. Lauderdale, Florida, USA, March 11–13, 2015, Prof. M. N. Hoda, General Chair, INDIACom – 2015 and Director, Bharati Vidyapeeth's Institute of Computer Applications and Management (BVICAM), New Delhi, India, pp. 967–972.

[21] Patterns Misconceptions, wiki.c2.com, verkkosivu, 2004. Saatavissa (viitattu 10.5.2018): http://wiki.c2.com/?PatternsMisconceptions

[22] PyYAML, Github, verkkosivu, 2006. Saatavissa (viitattu 20.10.2018): https://github.com/yaml/pyyaml

[23] D. Riehle, Framework Design A Role Modeling Approach, dissertation, Universität Hamburg, Publication 13509, 2000, 212 p. Saatavissa: http://dirkriehle.com/computer-science/research/dissertation/diss-a4.pdf

[24] T. Salmi, J. Kallioniemi, J. Loehr, Kaira-core: Data extraction from Finnish person catalogues, Github, tietokoneohjelma, 2017. Saatavissa (viitattu 27.10.2017): https://github.com/Learning-from-our-past/kaira-core